Security Audit Report

Fx Shareable Rebalance Pool

by AladdinDAO

SECBIT

January 18, 2024

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 supports owners in sharing their holdings of veFXN tokens to enhance stakers' returns. SECBIT Labs conducted an audit from January 2 to January 18, 2024, 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
Gas Optimization	4.3.1 The function `_updateBoostCheckpoint()` should be split into different modules to reduce the gas cost for the caller.	Info	Fixed
Design & Implementation	4.3.2 Consecutive calls to the `acceptSharedVote()` function may unexpectedly revoke the owner's share behavior.	Info	Fixed
Design & Implementation	4.3.3 The quantity of veFXN tokens held by the owner is not updated promptly.	Medium	Fixed
Design & Implementation	4.3.4 There is an error in assigning the parameter `prevWeekTs`.	Medium	Fixed
Design & Implementation	4.3.5 In specific scenarios, the quantity of veFXN tokens obtained through the `totalSupply()` function does not align with the actual situation.	Medium	Fixed
Gas Optimization	4.3.6 Optimizing the code structure of the `checkpoint()` function to save gas.	Info	Fixed
Design & Implementation	4.3.7 The value of the parameter `_supply[week].epoch` may not align with the actual situation.	Medium	Fixed
Gas Optimization	4.3.8 Discussion on the parameter `voteOwnerHistoryBalances`.	Info	Fixed
Design & Implementation	4.3.9 Add input parameter validation for `timestamp` in the `checkpoint()` function to prevent the results that do not align with the actual situation.	Info	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the shareable rebalance pool of the f(x) protocol is shown below:

- Smart contract code
    - initial review commit
        - caac0e4
    - final review commit
        - 69378f5

2.2 Contract List

The following content shows the contracts included in the shareable rebalance pool of f(x) protocol, which the SECBIT team audits:

Name	Lines	Description
ShareableRebalancePool.sol	521	Support the sharing of veFXN tokens held by the owner to enhance the earnings of stakers.
VotingEscrowBoost.sol	267	Holders of veFXN tokens can use the boost mechanism in this contract to increase their earnings.
VotingEscrowHelper.sol	174	Auxiliary contract to record the quantity of veFXN tokens, assisting high-frequency users in saving gas.

Notice: This audit specifically focuses on the modified portions of the VotingEscrowBoost.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 shareable rebalance pool of f(x) protocol are Governance Account and Common Account.

- Governance Account
    - Description

      Contract Administrator
    - Authority
        - Update protocol parameter
        - Transfer ownership
        - Share the veFXN tokens held by the owner with the stakers
    - 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
        - Accept / Reject shared vote
        - 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. Compared to the old version, the new mechanism supports the owner sharing the held veFXN tokens, aiding stakers in increasing their FXN token earnings. 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:

ShareableRebalancePool

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. Rebalance Pool depositors acquire collateral (in stETH) from the liquidated positions at a considerable discount as a reward. 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.

- `toggleVoteSharing()`

  The owner needs to utilize this function to allow stakers to use their veFXN tokens and enhance their FXN token earnings.

- `acceptSharedVote()` / `rejectSharedVote()`

  Stakers have the option to accept or reject the owner's vote sharing.

VotingEscrowHelper

To reduce the high gas consumption incurred by frequent queries of veFXN token quantities by high-frequency users, this contract will record the quantity of veFXN tokens at each integer multiple of `WEEK` timestamps. Subsequent users can directly use these values without needing a binary search, significantly lowering user gas consumption.

- `checkpoint()`

  Snapshot the state of some user.
- `totalSupply()`

  Return the ve total supply at some specific time point.
- `balanceOf()`

  Return the ve balance of some user at some specific time point.

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 function `_updateBoostCheckpoint()` should be split into different modules to reduce the caller's gas cost.

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

Location

ShareableRebalancePool.sol#L594-L637

Description

The function `_updateBoostCheckpoint()` has two functionalities: Firstly, it updates the principal amount of fToken held in custody by the owner in the event of the owner being active, preparing for potential liquidation; Secondly, it updates the staker's boost ratio, and records it in `boostCheckpoint[_account]`. Combining these two functionalities within a single function `_updateBoostCheckpoint()` might seem less logical.

The function `_updateBoostCheckpoint()` will be utilized within the `deposit()`, `withdraw()`, `toggleVoteSharing()`, `acceptSharedVote()`, and `rejectSharedVote()` functions. Taking the `deposit()` function as an example for detailed explanation, similar logic applies to other functions such as `withdraw()`. Assuming a staker has obtained authorization from the owner and can use the owner's veFXN tokens to increase their boost ratio.

When he uses the `deposit()` function to deposit fTokens into the protocol, the following operations will be performed:

(1). Invoke the `_checkpoint()` function to update the staker's earnings, the principal amount of fTokens held by the staker, the principal amount of fTokens held in custody by the owner (considering liquidation), and the staker's latest information in the `boostCheckpoint[]`. Note that the `_updateBoostCheckpoint()` function is called at the end of the `_checkpoint()` function.

(2). Update the principal amount of fTokens deposited by the staker, the total principal amount deposited by all users in the protocol, and the principal amount of fTokens held in custody by the owner.

(3). Once again, call the `_updateBoostCheckpoint()` function to update the principal amount of fTokens held in custody by the owner (considering liquidation), and the latest information in the `boostCheckpoint[]` for the user.

When the staker calls `deposit()` function, there are two consecutive calls to the `_updateBoostCheckpoint()` function, leading to unnecessary code duplication and increased gas consumption for the staker. Specifically, the first call to `_updateBoostCheckpoint()` in the `deposit()` function (within the `_checkpoint()` function) is necessary for updating the principal amount of fTokens held in custody by the owner, considering potential liquidation scenarios. However, the subsequent update of the staker's boost ratio and recording it in the `boostCheckpoint[]` array is unnecessary, as these values will be overwritten in the second call to `_updateBoostCheckpoint()`.

In the second call to `_updateBoostCheckpoint()` in the `deposit()` function, updating the principal amount of fTokens held in custody by the owner is unnecessary, as already considered in the first call. However, updating the staker's boost ratio and recording it in the `boostCheckpoint[]` is necessary due to changes in the staker's funds.

Considering this, it might be worthwhile to explore the possibility of splitting the functionality of the `_updateBoostCheckpoint()` function into two separate functions to avoid unnecessary code duplication.

```
function deposit(uint256 _amount, address _receiver) external override {
    if (hasRole(VE_SHARING_ROLE, _receiver)) revert
ErrorVoteOwnerCannotStake();

    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.
    // @audit this function also invokes the _updateBoostCheckpoint()
function.
    _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);

    // @note after checkpoint, the voteOwnerBalances are correct.
    address _owner = getStakerVoteOwner[_receiver];
    if (_owner != address(0)) {
      voteOwnerBalances[_owner].amount += uint104(_amount);
    }

    // this is already updated in `_checkpoint(_receiver)`.
    // _balance.updateAt = uint40(block.timestamp);

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

    // update boost checkpoint at last
    // @audit the _updateBoostCheckpoint() function is invoked once
again.
    _updateBoostCheckpoint(_receiver, _balance, _supply);

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

function _updateBoostCheckpoint(
    address _account,
    TokenBalance memory _balance,
    TokenBalance memory _supply
  ) internal {
    // update `voteOwnerBalances[_owner]` to latest epoch and record
history value
    address _owner = getStakerVoteOwner[_account];
    address _veHolder = _owner == address(0) ? _account : _owner;
    TokenBalance memory _ownerBalance;
    if (_owner != address(0)) {
      _ownerBalance = voteOwnerBalances[_owner];
      // @note the value of `_ownerBalance.updateAt` should never be
zero, since it will be updated before this call.
      uint256 prevWeekTs = ((uint256(_ownerBalance.updateAt) + WEEK - 1)
/ WEEK) * WEEK;
      _ownerBalance.amount = uint104(
        _getCompoundedBalance(_ownerBalance.amount,
_ownerBalance.product, _supply.product)
      );
      _ownerBalance.product = _supply.product;
      _ownerBalance.updateAt = uint40(block.timestamp);
      voteOwnerBalances[_owner] = _ownerBalance;

      // Normally, `prevWeekTs` equals to `nextWeekTs` so we will only
sstore 1 time in most of the time.
      //
      // When `prevWeekTs < nextWeekTs`, there are some extreme situation
that liquidation happens between
      // `_ownerBalance.updateAt` and `prevWeekTs`, also some time
between `prevWeekTs` and `block.timestamp`.
      // Then we cannot calculate the amount at `prevWeekTs` correctly.
Since the situation rarely happens,
      // it is ok to use `_ownerBalance.amount` only.
      uint256 nextWeekTs = ((block.timestamp + WEEK - 1) / WEEK) * WEEK;
      while (prevWeekTs <= nextWeekTs) {
        voteOwnerHistoryBalances[_owner][prevWeekTs] =
_ownerBalance.amount;
        prevWeekTs += 1;
      }
    } else {
      _ownerBalance = _balance;
    }

    uint256 _ratio = _computeBoostRatio(
      _ownerBalance.amount,
      _balance.amount,
      _supply.amount,
      IVotingEscrow(ve).balanceOf(_veHolder),
      IVotingEscrow(ve).totalSupply()
    );
    boostCheckpoint[_account] = BoostCheckpoint(uint64(_ratio),
uint64(numTotalSupplyHistory));
  }
```

Status

The development team has adopted our suggestion and adjusted the logic of the `_updateBoostCheckpoint()` function in commit 2c9317c.

4.3.2 Consecutive calls to the `acceptSharedVote()` function may unexpectedly revoke the owner's share behavior.

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

Location

ShareableRebalancePool.sol#L407

Description

The staker must accept the owner's share of veFXN tokens through the `acceptSharedVote()` function. Consider the following scenario: the staker calls the `acceptSharedVote()` function twice, passing the same owner address, resulting in the revocation of the owner's share. This situation is inconsistent with the original intent of the function and should be avoided.

```
function acceptSharedVote(address _newOwner) external override {
    address _staker = _msgSender();
    if (!isStakerAllowed[_newOwner][_staker]) {
      revert ErrorVoteShareNotAllowed();
    }

    address _oldOwner = getStakerVoteOwner[_staker];
    if (_oldOwner != address(0)) {
      _revokeVoteSharing(_oldOwner, _staker);
    } else {
      // @note after checkpoint, the epoch of `_balances[_staker]` and
`voteOwnerBalances[_oldOwner]`
      // are on the latest epoch, we can safely to do add or subtract.
      _checkpoint(_staker);
    }

    // update voteOwnerBalances[_newOwner] to latest epoch
    TokenBalance memory _balance = _balances[_staker];
    TokenBalance memory _supply = _totalSupply;
    TokenBalance memory _ownerBalance = voteOwnerBalances[_newOwner];
    _ownerBalance.amount =
uint104(_getCompoundedBalance(_ownerBalance.amount,
_ownerBalance.product, _supply.product));
    _ownerBalance.amount += _balance.amount;
    _ownerBalance.product = _supply.product;
    _ownerBalance.updateAt = uint40(block.timestamp);

    voteOwnerBalances[_newOwner] = _ownerBalance;
    getStakerVoteOwner[_staker] = _newOwner;

    _updateBoostCheckpoint(_staker, _balance, _supply);

    emit AcceptSharedVote(_staker, address(0), _newOwner);
  }
```

Suggestion

Enhance the assessment of the parameter `_newOwner` by incorporating a condition that triggers a revert if `_newOwner == _oldOwner`.

```
//@audit add the following code within the IFxShareableRebalancePool
interface contract
error ErrorRepeatAcceptSharedVote();

function acceptSharedVote(address _newOwner) external override {
    address _staker = _msgSender();
    if (!isStakerAllowed[_newOwner][_staker]) {
      revert ErrorVoteShareNotAllowed();
    }

    address _oldOwner = getStakerVoteOwner[_staker];

    //@audit add the following restriction
    if (_oldOwner == _newOwner) {
      revert ErrorRepeatAcceptSharedVote();
    }

    if (_oldOwner != address(0)) {
      _revokeVoteSharing(_oldOwner, _staker);
    } else {
      // @note after checkpoint, the epoch of `_balances[_staker]` and
`voteOwnerBalances[_oldOwner]`
      // are on the latest epoch, we can safely to do add or subtract.
      _checkpoint(_staker);
    }

   ......
  }
```

Status

The development team has adopted our suggestion and addressed this issue in commit 2c9317c.

4.3.3 The quantity of veFXN tokens held by the owner is not updated promptly.

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

Location

ShareableRebalancePool.sol#L492

Description

Due to the restriction that the `_account` parameter passed to the `_checkpoint()` function must be the address of a staker, the quantity of veFXN tokens held by the owner cannot be updated. The only method to update the veFXN token quantity held by the owner is by externally calling the `veHelper.checkpoint(owner)` function. It directly impacts the calculation of the boost ratio for stakers, potentially leading to inaccuracies in calculating FXN token earnings for stakers.

```
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 the veFXN tokens held by the owner can not be updated
    IVotingEscrowHelper(veHelper).checkpoint(_account);

    MultipleRewardCompoundingAccumulator._checkpoint(_account);

    if (_account != address(0)) {
      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;

      _updateBoostCheckpoint(_account, _balance, _supply);
    }
  }
```

Status

The development team addressed this issue in commit 2c9317c.

4.3.4 There is an error in the parameter `prevWeekTs` assignment.

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

Location

ShareableRebalancePool.sol#L623

Description

The parameters `prevWeekTs` and `nextWeekTs` are initialized as multiples of weeks. Therefore, within the while loop, the increment of the `prevWeekTs` parameter should also be in multiple weeks.

```
function _updateBoostCheckpoint(
    address _account,
    TokenBalance memory _balance,
    TokenBalance memory _supply
```

```
  ) internal {
    // update `voteOwnerBalances[_owner]` to latest epoch and record
history value
    address _owner = getStakerVoteOwner[_account];
    address _veHolder = _owner == address(0) ? _account : _owner;
    TokenBalance memory _ownerBalance;
    if (_owner != address(0)) {
      _ownerBalance = voteOwnerBalances[_owner];
      // @note the value of `_ownerBalance.updateAt` should never be
zero, since it will be updated before this call.
      // @audit certainly, `prevWeekTs` is guaranteed to be a multiple of
weeks.
      uint256 prevWeekTs = ((uint256(_ownerBalance.updateAt) + WEEK - 1)
/ WEEK) * WEEK;
      _ownerBalance.amount = uint104(
        _getCompoundedBalance(_ownerBalance.amount,
_ownerBalance.product, _supply.product)
      );
      _ownerBalance.product = _supply.product;
      _ownerBalance.updateAt = uint40(block.timestamp);
      voteOwnerBalances[_owner] = _ownerBalance;

      // Normally, `prevWeekTs` equals to `nextWeekTs` so we will only
sstore 1 time in most of the time.
      //
      // When `prevWeekTs < nextWeekTs`, there are some extreme situation
that liquidation happens between
      // `_ownerBalance.updateAt` and `prevWeekTs`, also some time
between `prevWeekTs` and `block.timestamp`.
      // Then we cannot calculate the amount at `prevWeekTs` correctly.
Since the situation rarely happens,
      // it is ok to use `_ownerBalance.amount` only.

      // @audit `nextWeekTs` is also a multiple of weeks.
      uint256 nextWeekTs = ((block.timestamp + WEEK - 1) / WEEK) * WEEK;
      while (prevWeekTs <= nextWeekTs) {
        voteOwnerHistoryBalances[_owner][prevWeekTs] =
_ownerBalance.amount;

        //@audit here, `WEEK` should be added instead of 1.
        prevWeekTs += 1;
      }
    } else {
      _ownerBalance = _balance;
    }
    ......
  }
```

Status

The development team addressed this issue in commit 2c9317c.

4.3.5 In specific scenarios, the quantity of veFXN tokens obtained through the `totalSupply()` function does not align with the actual situation.

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

Location

VotingEscrowHelper.sol#L80

Description

The function `totalsupply()` is utilized to retrieve the total quantity of veFXN tokens at a specific timestamp. In certain scenarios, discrepancies may arise between the quantity of veFXN tokens obtained through this function and directly querying the veFXN token contract.

Since the data for this function is sourced from the veFXN token contract, let's first analyze the method of obtaining veFXN tokens under the veFXN token contract.

The veFXN token contract doesn't have a direct interface to fetch veFXN tokens at a specific timestamp. To address this, it incorporates the `totalSupplyAt()` function. Users can specify a particular block height, and it provides the total veFXN tokens at the corresponding epoch (timestamp). In detail, when a user provides a specific block height, the function employs binary search to find the nearest epoch with a height not exceeding `_block`. Using this epoch's data as a reference, and considering `dt` as an offset, it approximates the timestamp corresponding to the block height `_block`. Subsequently, it calls `supply_at()` to calculate the veFXN token quantity at the block height. Due to the inability to directly fetch the timestamp corresponding to a specified block height from the chain, the code resorts to an approximation method to obtain the timestamp. This approximation introduces a potential discrepancy between the calculated time and the actual timestamp corresponding to the block.

Certainly, let's further analyze the approach of obtaining the quantity of veFXN tokens at a specified timestamp under the `VotingEscrowHelper.sol` contract.

The `VotingEscrowHelper.sol` contract provides the `totalSupply()` function, allowing users to directly obtain the corresponding veFXN token total at a specified `timestamp`. In detail, the function utilizes binary search to retrieve the nearest epoch data with a timestamp not exceeding the specified `timestamp`. Subsequently, it calculates the corresponding quantity of veFXN tokens based on this epoch data.

Comparing the two approaches for calculating the veFXN token at a specified moment reveals two issues:

(1). Due to the continuous nature of time and the discrete recording of time in block heights, a one-to-one correspondence between time and block height is impossible. The `totalSupply()` function under the `VotingEscrowHelper.sol` contract can retrieve the total amount of veFXN tokens at any given moment. However, using the veFXN token contract only provides the quantity of veFXN tokens corresponding to a specific block height. From this perspective, the `totalSupply()` function in the `VotingEscrowHelper.sol` contract is more comprehensive. (It's ok here.)

(2). If one intends to calculate the veFXN tokens corresponding to a specific block height using the `totalSupply()` function under the `VotingEscrowHelper.sol` contract, in certain situations (when the input `timestamp` parameter is not a multiple of a week), discrepancies may arise between the return values of the `totalSupply()` function and those obtained directly from the `totalSupplyAt()` function under the veFXN token contract. This issue is attributed to rounding errors in calculating the `dt` parameter within the `totalSupplyAt()` function. The specific data explaining the issue is as follows:

Assuming we want to inquire about the total supply of veFXN tokens corresponding to the block height 18588600, values obtained from two different functions are calculated:

- The quantity of veFXN tokens obtained through the `totalSupplyAt()` function of the veFXN token contract is 63,734,528,770,233,930,511,551.
- The calculation process for veFXN tokens through the `totalSupply()` function under the `VotingEscrowHelper.sol` contract is as follows: Firstly, based on the block height of 18588600, the corresponding timestamp is found to be 1,700,187,839. The veFXN token quantity calculated from this timestamp is 63,734,531,434,818,843,010,923.

At this point, it is observed that the quantities of veFXN tokens obtained from these two functions are not the same. Further investigation is needed to confirm and address this issue.

Considering the actual usage parameters in the protocol, the `totalSupply()` function under the `VotingEscrowHelper.sol` contract is only used in the `_getBoostRatio()` function of the ShareableRebalancePool.sol contract. The parameter `_nowTs` passed to it is always a multiple of weeks. It conveniently avoids the rounding error introduced by the `dt` parameter, as in this case, `dt == 0`. In such situations, the quantities of veFXN tokens calculated by the two functions are identical.

```
@external
@view
def totalSupplyAt(_block: uint256) -> uint256:
    """
    @notice Calculate total voting power at some point in the past
    @param _block Block to calculate the total voting power at
    @return Total voting power at `_block`
    """
    assert _block <= block.number
    _epoch: uint256 = self.epoch
    // @audit query the most recent epoch information for a time not
exceeding the one corresponding to the _block, and calculate the veFXN
token quantity for the block height based on this information.
    target_epoch: uint256 = self.find_block_epoch(_block, _epoch)

    point: Point = self.point_history[target_epoch]
    dt: uint256 = 0
    if target_epoch < _epoch:
        point_next: Point = self.point_history[target_epoch + 1]
        if point.blk != point_next.blk:
            // @audit compute the time offset. Here, approximate
calculation of the time corresponding to the block height is performed,
with potential rounding errors.
            dt = (_block - point.blk) * (point_next.ts - point.ts) /
(point_next.blk - point.blk)
    else:
        if point.blk != block.number:
            dt = (_block - point.blk) * (block.timestamp - point.ts) /
(block.number - point.blk)
    # Now dt contains info on how far are we beyond point

    return self.supply_at(point, point.ts + dt)

def supply_at(point: Point, t: uint256) -> uint256:
    """
    @notice Calculate total voting power at some point in the past
    @param point The point (bias/slope) to start search from
    @param t Time to calculate the total voting power at
    @return Total voting power at that time
    """
    last_point: Point = point
    t_i: uint256 = (last_point.ts / WEEK) * WEEK
    for i in range(255):
        t_i += WEEK
        d_slope: int128 = 0
        if t_i > t:
            t_i = t
        else:
            d_slope = self.slope_changes[t_i]
        last_point.bias -= last_point.slope * convert(t_i -
last_point.ts, int128)
        if t_i == t:
            break
        last_point.slope += d_slope
        last_point.ts = t_i

    if last_point.bias < 0:
        last_point.bias = 0
    return convert(last_point.bias, uint256)
```

```
function totalSupply(uint256 timestamp) external view override returns
(uint256) {
    if (timestamp < start) return 0;

    uint256 week = (timestamp / WEEK) * WEEK;
    Balance memory prevSupply = _supply[week];
    IVotingEscrow.Point memory point;
    if (prevSupply.epoch > 0) {
      if (week == timestamp) return prevSupply.value;
      Balance memory nextSupply = _supply[week + WEEK];
      uint256 nextEpoch = nextSupply.epoch;
      if (nextEpoch == 0) nextEpoch = IVotingEscrow(ve).epoch();
      (, point) = _binarySearchSupplyPoint(timestamp, prevSupply.epoch,
nextEpoch);
    } else {
       //@audit in the case of a future time being provided as input, the
query value here may result in inaccuracies.
      (, point) = _binarySearchSupplyPoint(timestamp, 1,
IVotingEscrow(ve).epoch());
    }
    return _computeValue(point, timestamp);
  }

function _binarySearchSupplyPoint(
  uint256 timestamp,
  uint256 startEpoch,
  uint256 endEpoch
) internal view returns (uint256 epoch, IVotingEscrow.Point memory point)
{
  unchecked {
    while (startEpoch < endEpoch) {
      uint256 mid = (startEpoch + endEpoch + 1) / 2;
      IVotingEscrow.Point memory p =
IVotingEscrow(ve).point_history(mid);
      if (p.ts <= timestamp) {
        startEpoch = mid;
        point = p;
      } else {
        endEpoch = mid - 1;
      }
    }
  }
  epoch = startEpoch;
  // in case, the `p.ts <= timestamp` never hit in the binary search
  if (point.ts == 0) {
    point = IVotingEscrow(ve).point_history(epoch);
  }
}

function _computeValue(IVotingEscrow.Point memory point, uint256
timestamp) internal pure returns (uint256) {
    int256 bias = point.bias - point.slope * int256(timestamp -
point.ts);
    if (bias < 0) bias = 0; // the lock has expired, only happens when it
is the last point

    return uint256(bias);
}
```

In addition to the previously mentioned issues, consider a new scenario where the veFXN token contract has not been invoked for over two weeks, resulting in the epoch going without updates for over two weeks. In this case, errors may occur in calculating the veFXN token quantity when using the `totalSupply()` function under the `VotingEscrowHelper.sol` contract. This error becomes more pronounced when estimating the veFXN token quantity at some future moment.

Assuming there have been no user calls to the veFXN token contract for two weeks, let's first examine the `totalSupply()` function under the veFXN contract. When calculating the veFXN token quantity, it considers the possibility of users' locks expiring and adjusts the value of the slope parameter accordingly. However, the `totalSupply()` function under the

VotingEscrowHelper.sol contract does not account for this scenario during its calculation.

```
def totalSupply(t: uint256 = block.timestamp) -> uint256:
    """
    @notice Calculate total voting power
    @dev Adheres to the ERC20 `totalSupply` interface for Aragon
compatibility
    @return Total voting power
    """
    _epoch: uint256 = self.epoch
    last_point: Point = self.point_history[_epoch]
    return self.supply_at(last_point, t)

def supply_at(point: Point, t: uint256) -> uint256:
    """
    @notice Calculate total voting power at some point in the past
    @param point The point (bias/slope) to start search from
    @param t Time to calculate the total voting power at
    @return Total voting power at that time
    """
    last_point: Point = point
    t_i: uint256 = (last_point.ts / WEEK) * WEEK
    for i in range(255):
        t_i += WEEK
        d_slope: int128 = 0
        if t_i > t:
            t_i = t
        else:
            //@audit the slope value will be corrected, as there may be
users whose lock has expired.
            d_slope = self.slope_changes[t_i]
        last_point.bias -= last_point.slope * convert(t_i -
last_point.ts, int128)
        if t_i == t:
            break
        last_point.slope += d_slope
        last_point.ts = t_i

    if last_point.bias < 0:
        last_point.bias = 0
    return convert(last_point.bias, uint256)
```

In summary, there are two issues in the implementation of the totalSupply() function:

- When the input timestamp is not a multiple of a week, the veFXN token quantity returned by this function is inconsistent with the veFXN token quantity obtained directly from the veFXN token contract (assuming that the timestamp corresponds exactly to a block height).
- In the scenario where there have been no user calls to the veFXN token contract for two weeks, the veFXN token quantity read by the `totalSupply()` function under the `VotingEscrowHelper.sol` contract will be incorrect.

Status

The development team has verified this issue. The current code still utilizes the direct input of time to calculate the quantity of veFXN tokens. This does not impact the normal operation of the protocol. Additionally, the development team addressed the issue of not considering slope changes when calculating the total veFXN token quantity in commit 2c9317c.

4.3.6 Optimizing the code structure of the `checkpoint()` function to save gas.

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

Location

VotingEscrowHelper.sol#L148

VotingEscrowHelper.sol#L136

Description

In practice, there may be situations where the veFXN token contract remains unchanged for two weeks. It can result in the `week` parameter within the `checkpoint()` function exceeding the time corresponding to the latest epoch for veFXN tokens. In such cases, there is no need to invoke the `_binarySearchSupplyPoint()` function for binary search; instead, utilizing the latest epoch-related data can save gas.

Additionally, under the `ShareableRebalancePool.sol` contract, the `checkpoint()` function is called to update the balances of veFXN tokens for user accounts. In cases where the account is the zero address (during liquidation), there is no need to continue calculating the balance for that account. Modifying the code to skip this logic and save gas is recommended.

```
function checkpoint(address account) external override {
    // checkpoint supply
    uint256 week = (block.timestamp / WEEK) * WEEK;
    Balance memory nowSupply = _supply[week];
    if (nowSupply.epoch == 0) {
      Balance memory prevSupply = _supply[week - WEEK];
      uint256 epoch;
      IVotingEscrow.Point memory point;
      if (prevSupply.epoch == 0) {
         // @audit in cases where the veFXN contract is not updated
promptly, the updated `week` here may exceed the time corresponding to
the latest epoch. In such situations, it is possible to directly use the
latest epoch value.
        (epoch, point) = _binarySearchSupplyPoint(week, 1,
IVotingEscrow(ve).epoch());
      } else {
        (epoch, point) = _binarySearchSupplyPoint(week, prevSupply.epoch,
IVotingEscrow(ve).epoch());
      }

      nowSupply.value = uint128(_computeValue(point, week));
      nowSupply.epoch = uint128(epoch);
      _supply[week] = nowSupply;
    }

    // checkpoint balance
    // @audit when the `account` is the zero address, the following
operations can be skipped.
    uint256 userPointEpoch = IVotingEscrow(ve).user_point_epoch(account);

    if (userPointEpoch == 0) return;

    Balance memory nowBalance = _balances[account][week];
    if (nowBalance.epoch == 0) {
      Balance memory prevBalance = _balances[account][week - WEEK];
      uint256 epoch;
      IVotingEscrow.Point memory point;
      if (prevBalance.epoch == 0) {
        (epoch, point) = _binarySearchBalancePoint(account, week, 1,
userPointEpoch);
      } else {
        (epoch, point) = _binarySearchBalancePoint(account, week,
prevBalance.epoch, userPointEpoch);
      }

      nowBalance.value = uint128(_computeValue(point, week));
      nowBalance.epoch = uint128(epoch);
```

```
        _balances[account][week] = nowBalance;
      }
    }
```

Status

The development team has adopted our suggestions and addressed the issue in commit 2c9317c.

4.3.7 The value of the parameter `_supply[week].epoch` may not align with the actual situation.

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

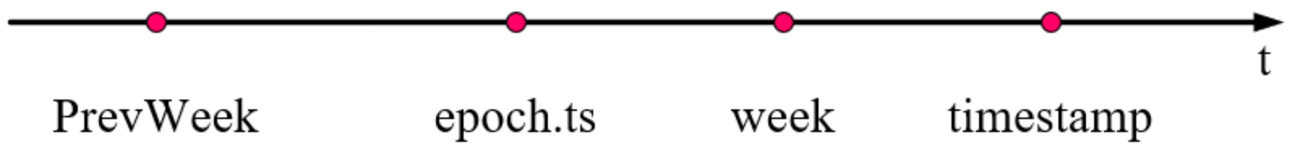
Location

VotingEscrowHelper.sol#L142

Description

The function `checkpoint()` is utilized to update the veFXN token quantities under the veFXN token contract and for the user at the time of `week`. In this scenario, there might be a discrepancy between the veFXN token quantity recorded by `_supply[week]` and the values recorded under the `epoch` and the veFXN token contract.

Consider the following scenario: Assuming the current veFXN token contract has not updated the epoch state, resulting in the following relationships among `timestamp` (current timestamp), `week` (time obtained by rounding down the timestamp weekly), `epoch.ts` (time corresponding to the global latest epoch value of the veFXN token contract), and `PreWeek` (timestamp of the previous week).



In this scenario, if a user calls the `checkpoint()` function to update the `_supply[week]` parameter, the following logic will be executed:

- Utilizing the `_binarySearchSupplyPoint()` function for binary search to obtain the `epoch`. At this moment, since `week` is greater than the time corresponding to the latest `IVotingEscrow(ve).epoch()` in the veFXN token contract (i.e., the `epoch.ts` in the above diagram), the result of the binary search will land on the boundary value, which is the epoch at the `epoch.ts` moment.
- Calling the `_computeValue()` function to calculate the total veFXN tokens at the `week`. It's essential to note that, at this point, parameters related to the `epoch.ts` moment are used as a baseline to calculate the total veFXN tokens at the `week`.
- Incorrectly recording the `epoch` value at the `epoch.ts` moment into the `_supply[week]` parameter directly.

In reality, the `epoch` value at the `_supply[week]` moment should be one greater than the `epoch` value at the `epoch.ts` moment for the corresponding `week`.

```
function checkpoint(address account) external override {
    // checkpoint supply
    uint256 week = (block.timestamp / WEEK) * WEEK;
    Balance memory nowSupply = _supply[week];
    if (nowSupply.epoch == 0) {
      Balance memory prevSupply = _supply[week - WEEK];
      uint256 epoch;
      IVotingEscrow.Point memory point;
      if (prevSupply.epoch == 0) {
        (epoch, point) = _binarySearchSupplyPoint(week, 1,
IVotingEscrow(ve).epoch());
      } else {
          //@audit due to the fact that the `week` time is greater than
the time corresponding to `IVotingEscrow(ve).epoch()`, the epoch obtained
through binary search will ultimately land on the boundary value of
`IVotingEscrow(ve).epoch()`.
        (epoch, point) = _binarySearchSupplyPoint(week, prevSupply.epoch,
IVotingEscrow(ve).epoch());
      }

      nowSupply.value = uint128(_computeValue(point, week));
      //@audit the recorded epoch value here may not be aligned with the
week.
      nowSupply.epoch = uint128(epoch);
      _supply[week] = nowSupply;
    }

   ......
  }
```

Status

The development team has decided to address the issue of delayed updates in the veFXN token contract state by calling the `checkpoint()` function under the veFXN token contract before updating the parameters of the `VotingEscrowHelper.sol` contract. The relevant fix code can be found in commit 2c9317c.

4.3.8 Discussion on the parameter `voteOwnerHistoryBalances`.

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

Location

ShareableRebalancePool.sol#L606

Description

The parameter `voteOwnerHistoryBalances` records the principal quantity escrowed by the `owner` address at multiples of `WEEK`. It gets updated in the functions `_updateVoteOwnerBalance()` and `_updateBoostCheckpoint()`. It's worth noting that this parameter records the principal quantity escrowed by the `owner` address at the future nearest multiples of `WEEK`, even if this quantity might be updated in the future. There are two considerations regarding whether it is necessary to record the principal quantity at the future nearest multiples of `WEEK`:

In scenario one, it is unnecessary to record the principal amount at future time points that are multiples of `WEEK` for calculating the boost ratio. Such recording operations would consume the caller's gas unnecessarily. From this perspective, it is feasible to eliminate the recording of the principal amount at future time points that are multiples of `WEEK`. It requires concurrent modifications to the `_updateVoteOwnerBalance()` function (excluding the equal sign) and relevant content within the `_updateBoostCheckpoint()` function.

In scenario two, if we consider that the `voteOwnerHistoryBalances` parameter allows real-time querying of the principal amount under custody at future time points that are multiples of `WEEK` (although this value may still be subject to updates), it is sufficient to modify only the `_updateVoteOwnerBalance()` function (excluding the equal sign). It is because even if the principal amount at future time points that are multiples of `WEEK` is updated within the `_updateVoteOwnerBalance()` function, it will be overwritten in the subsequent invocation of the `_updateBoostCheckpoint()` function.

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

    if (_account != address(0)) {
      TokenBalance memory _supply = _totalSupply;
      TokenBalance memory _balance = _updateUserBalance(_account,
_supply);
      TokenBalance memory _ownerBalance = _updateVoteOwnerBalance(_owner,
_supply);
      _updateBoostCheckpoint(_account, _owner, _balance, _ownerBalance,
_supply);
    }
  }

function acceptSharedVote(address _newOwner) external override {
 ......

  // update boost ratio for staker.
  TokenBalance memory _balance = _balances[_staker];
  TokenBalance memory _supply = _totalSupply;
  TokenBalance memory _ownerBalance = _updateVoteOwnerBalance(_newOwner,
_supply);
   _ownerBalance.amount += _balance.amount;
   _updateBoostCheckpoint(_staker, _newOwner, _balance, _ownerBalance,
_supply);

   emit AcceptSharedVote(_staker, address(0), _newOwner);
 }

 function _updateVoteOwnerBalance(address _owner, TokenBalance memory
_supply)
    internal
    virtual
    returns (TokenBalance memory _balance)
  {
    // update `voteOwnerBalances[_owner]` to latest epoch and record
history value
    if (_owner == address(0)) return _balance;
    _balance = voteOwnerBalances[_owner];
    // @note the value of `_ownerBalance.updateAt` should never be zero,
since it will be updated before this call.
    uint256 prevWeekTs = _getWeekTs(_balance.updateAt);
    _balance.amount = uint104(_getCompoundedBalance(_balance.amount,
_balance.product, _supply.product));
    _balance.product = _supply.product;
    _balance.updateAt = uint40(block.timestamp);
    // @audit this operation is redundant, as the subsequent
`_updateBoostCheckpoint()` will once again update this value.
    voteOwnerBalances[_owner] = _balance;

    // Normally, `prevWeekTs` equals to `nextWeekTs` so we will only
sstore 1 time in most of the time.
    //
    // When `prevWeekTs < nextWeekTs`, there are some extreme situation
that liquidation happens between
    // `_ownerBalance.updateAt` and `prevWeekTs`, also some time between
`prevWeekTs` and `block.timestamp`.
    // Then we cannot calculate the amount at `prevWeekTs` correctly.
Since the situation rarely happens,
    // it is ok to use `_ownerBalance.amount` only.
    // @audit the `nextWeekTs` is not less than the current timestamp and
is a multiple of `WEEK`.
    uint256 nextWeekTs = _getWeekTs(block.timestamp);
    // @audit The equality condition is guaranteed to be satisfied. At
this point, the future nearest timestamp that is a multiple of `WEEK`
will be updated with the escrowed funds.
    while (prevWeekTs <= nextWeekTs) {
      voteOwnerHistoryBalances[_owner][prevWeekTs] = _balance.amount;
      prevWeekTs += WEEK;
    }
  }

function _updateBoostCheckpoint(
    address _account,
    address _owner,
    TokenBalance memory _balance,
    TokenBalance memory _ownerBalance,
    TokenBalance memory _supply
  ) internal {
    if (_owner == address(0)) {
      _ownerBalance = _balance;
      _owner = _account;
    } else {
      voteOwnerBalances[_owner] = _ownerBalance;
      // @audit this value is not be used in practice.
      uint256 nextWeekTs = _getWeekTs(block.timestamp);
      voteOwnerHistoryBalances[_owner][nextWeekTs] =
_ownerBalance.amount;
    }
    uint256 _ratio = _computeBoostRatio(
      _ownerBalance.amount,
      _balance.amount,
      _supply.amount,
      IVotingEscrow(ve).balanceOf(_owner),
      IVotingEscrow(ve).totalSupply()
    );
    boostCheckpoint[_account] = BoostCheckpoint(uint64(_ratio),
uint64(numTotalSupplyHistory));
  }
```

Suggestion

If following the above scenario two, the corresponding code can be modified as follows:

```
function _updateVoteOwnerBalance(address _owner, TokenBalance memory
_supply)
    internal
    virtual
    returns (TokenBalance memory _balance)
  {
    // update `voteOwnerBalances[_owner]` to latest epoch and record
history value
    if (_owner == address(0)) return _balance;
    _balance = voteOwnerBalances[_owner];
    // @note the value of `_ownerBalance.updateAt` should never be zero,
since it will be updated before this call.
    uint256 prevWeekTs = _getWeekTs(_balance.updateAt);
    _balance.amount = uint104(_getCompoundedBalance(_balance.amount,
_balance.product, _supply.product));
    _balance.product = _supply.product;
    _balance.updateAt = uint40(block.timestamp);

    // @audit remove the following code
    // voteOwnerBalances[_owner] = _balance;

    // Normally, `prevWeekTs` equals to `nextWeekTs` so we will only
sstore 1 time in most of the time.
    //
    // When `prevWeekTs < nextWeekTs`, there are some extreme situation
that liquidation happens between
    // `_ownerBalance.updateAt` and `prevWeekTs`, also some time between
`prevWeekTs` and `block.timestamp`.
    // Then we cannot calculate the amount at `prevWeekTs` correctly.
Since the situation rarely happens,
    // it is ok to use `_ownerBalance.amount` only.
    uint256 nextWeekTs = _getWeekTs(block.timestamp);
    // @audit remove equivalent conditions
    while (prevWeekTs < nextWeekTs) {
      voteOwnerHistoryBalances[_owner][prevWeekTs] = _balance.amount;
      prevWeekTs += WEEK;
    }
  }
```

Status

The development team has adopted our suggestions and removed the redundant code in commit 69378f5.

4.3.9 Add input parameter validation for `timestamp` in the `checkpoint()` function to prevent the results from aligning with the actual situation.

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

Location

VotingEscrowHelper.sol#L134

Description

The `checkpoint()` function should impose a lower limit on the `timestamp` parameter, ensuring that its value is not less than the `start` parameter's value. Otherwise, it may result in records that do not align with the actual situation.

```
  function checkpoint(address account, uint256 timestamp) external override
  {
      if (timestamp > block.timestamp) revert ErrorCheckpointFutureTime();

      // checkpoint supply
      uint256 week = (timestamp / WEEK) * WEEK;
      _checkpoint(account, week);
    }
```

Status

The development team has adopted our suggestions and added a restriction in commit 69378f5.

5. Conclusion

After auditing and analyzing the shareable rebalance pool of 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.