Security Audit Report

Rebalance Pool of f(x) Protocol by AladdinDAO

SECBIT

July 25, 2023

1. Introduction

AladdinDAO is a decentralized network aiming 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 introduces two new ETH derivative assets: fractional ETH (fETH) with stablecoin-like low volatility, and leveraged ETH (xETH), a leveraged long ETH perpetual token. The f(x) protocol utilizes a Rebalance Pool as its primary mechanism to liquidate collateralized debt positions that fall below the minimum collateralization ratio. SECBIT Labs conducted an audit from July 3 to July 25, 2023, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the RebalancePool contract has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising (see part 4 for details).

Type	Description	Level	Status
Design & Implementation	4.3.1 Discuss the internal implementation logic of the `refund()` function.	Info	Discussed
Design & Implementation	4.3.2 During contract initialization, it is essential to call the `updatePlatform()` function to update the `platform` parameter.	Info	Discussed
Comment Revise	4.3.3 The derivation of parameter `d[i]` in the code comments is incorrect.	Info	Fixed
Code Revise	4.3.4 The parameter name `UserRewardSnapsot` has a spelling mistake.	Info	Fixed
Design & Implementation	4.3.5 Parameter discussion for distributing extra rewards during protocol liquidation.	Info	Fixed
Design & Implementation	4.3.6 Discussion on the user fund withdrawal plan.	Info	Discussed
Design & Implementation	4.3.7 There is no capability to remove potentially expired reward tokens.	Info	Fixed
Design & Implementation	4.3.8 Discussion on the meaning of the parameter `_assetLossPerUnitStaked`.	Low	Fixed
Gas optimization	4.3.9 Add checks for the parameters `_initialDeposit` and `_initialUnlock` to save gas.	Info	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the RebalancePool contract of f(x) protocol is shown below:

- Smart contract code
    - initial review commit *c106519*
    - final review commit *dda0b4a*

2.2 Contract List

The following content shows the contracts included in the RebalancePool-related implementation, which the SECBIT team audits:

Name	Lines	Description
RebalanceWithBonusToken.sol	22	To provide a contract liquidation interface for any user and offer liquidation rewards to the caller.
stETHGateway.sol	53	Peripheral contract to convert the Ether deposited by users into stETH and participate in the f(x) protocol.
stETHTreasury.sol	69	Harvest pending stETH rewards to Rebalance Pool.
wstETHWrapper.sol	23	Auxiliary contract, providing the functionality for stETH and wstETH mutual exchange.
RebalancePool.sol	552	A Rebalance Pool to liquidate collateralized debt positions that fall below the minimum collateralization ratio.

3. Contract Analysis

This part describes code assessment details, including "role classification" and "functional analysis".

3.1 Role Classification

Three key roles in the RebalancePool contract are Governance Account and Common Account.

- Governance Account
    - Description

      Contract Administrator
    - Authority

        - Update the address of the liquidator
        - Update the address of the reward wrapper
        - Update the collateral ratio line for liquidation
        - Update the unlock duration after unlocking
        - Add or remove a new extra reward token
    - Method of Authorization

      The contract administrator is the contract's creator or authorized by transferring the governance account.

- Common Account
    - Description

      Users participate in the f(x) Protocol
    - Authority
        - Mint fETH / xETH with base tokens
        - Redeem base tokens with fETH / xETH tokens
        - Add base tokens to increase the collateralization ratio
        - Liquidate fETH tokens to increase the collateralization ratio
        - deposit and unlock assets in the Rebalance Pool
    - Method of Authorization

      No authorization required

3.2 Functional Analysis

The f(x) protocol uses a Rebalance Pool as its primary mechanism to liquidate collateralized debt positions that fall below the minimum collateralization ratio. The SECBIT team conducted a detailed audit of some of the contracts in the protocol. The critical functions of the contract are listed:

RebalancePool

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 below:

- `deposit()`

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

- `unlock()`

  Users can call this function to unlock their deposited assets, which will be locked and can only be withdrawn after a specified period.

- `withdrawUnlocked()`

  Users can use this function to withdraw their unlocked assets that have reached the expiration period.

- `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.

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 Discuss the internal implementation logic of the `refund()` function.

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

Description

The f(x) protocol adopts stETH as its `baseToken`, requiring users to convert their deposited Ether into stETH before it can be utilized by the protocol. In theory, one Ether and one stETH should have equal value. However, upon observing the contract code of stETH, it becomes evident that the amount of stETH obtained by exchanging one Ether is slightly less than 1e18. Here is the implementation code of stETH:

```
// @audit
https://etherscan.io/address/0x17144556fd3424edc8fc8a4c940b2d0493
6d17eb#code#F27#L166
/**
 * @return the amount of tokens owned by the `_account`.
 *
 * @dev Balances are dynamic and equal the `_account`'s share in
the amount of the
 * total Ether controlled by the protocol. See `sharesOf`.
 */

//@audit When obtaining user funds, the user's holdings will be
converted into an amount of stETH.
function balanceOf(address _account) external view returns
(uint256) {
    return getPooledEthByShares(_sharesOf(_account));
}

//@auidt
https://etherscan.io/address/0x17144556fd3424edc8fc8a4c940b2d0493
6d17eb#code#F27#L375
/**
 * @notice Moves `_amount` tokens from `_sender` to `_recipient`.
 * Emits a `Transfer` event.
 * Emits a `TransferShares` event.
 */
function _transfer(address _sender, address _recipient, uint256
_amount) internal {
    uint256 _sharesToTransfer = getSharesByPooledEth(_amount);
    _transferShares(_sender, _recipient, _sharesToTransfer);
    _emitTransferEvents(_sender, _recipient, _amount,
_sharesToTransfer);
}
```

Based on the provided information, it is apparent that, unlike a direct and equivalent conversion between Ether and weth quantities, when a user deposits Ether into the stETH contract, it is accounted for in the form of shares under the user's address. When reading the amount of stETH under a user's address, the shares are converted back.

The f(x) protocol handles the unused funds under the contract by calling the `refund()` function, which sends the remaining funds in the contract back to the user. The issue here is that when using the `balanceOf()` function to read the balance of a specific address and then send that balance back to the user, minor discrepancies may occur. This discrepancy could be due to the conversion of shares back to quantities and other factors in the process.

In terms of fund value, the minor discrepancy in shares has no significant impact on the current protocol users. However, considering future development work, developers should be cautious when combining the `balanceOf()` function and the `transfer()` function to transfer funds, as mentioned above.

```
function mintFToken(uint256 _minFTokenMinted) external payable
returns (uint256 _fTokenMinted) {
  ......
  _refund(stETH, msg.sender);
}

/// @notice Mint some xToken with some ETH.
/// @param _minXTokenMinted The minimum amount of xToken should
be received.
```

```
/// @return _xTokenMinted The amount of xToken should be
received.
function mintXToken(uint256 _minXTokenMinted) external payable
returns (uint256 _xTokenMinted) {
  ......
  _refund(stETH, msg.sender);
}

/// @notice Mint some xToken by add some ETH as collateral.
/// @param _minXTokenMinted The minimum amount of xToken should
be received.
/// @return _xTokenMinted The amount of xToken should be
received.
function addBaseToken(uint256 _minXTokenMinted) external payable
returns (uint256 _xTokenMinted) {
  ......

  _refund(stETH, msg.sender);
}

// @dev Internal function to refund extra token.
/// @param _token The address of token to refund.
/// @param _recipient The address of the token receiver.
function _refund(address _token, address _recipient) internal {
  uint256 _balance = IERC20(_token).balanceOf(address(this));

  IERC20(_token).safeTransfer(_recipient, _balance);
}
```

Status

The team has confirmed that it is a common phenomenon with stETH, and its current use does not pose security issues.

4.3.2 During contract initialization, it is essential to call the `updatePlatform()` function to update the `platform` parameter.

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

Description

The parameter `platform` is not initialized in the constructor, and there is no check for its value being 0 in the `harvest()` function. By default, the value of the `platform` parameter is 0, so calling the `harvest()` function in this state could mistakenly send the rewards to address 0x0, resulting in fund loss. However, it will eliminate after the administrator actively calls the `updatePlatform()` function.

```
/// @notice The address platform contract.
address public platform;

/// @notice Harvest pending stETH rewards to stability pool.
function harvest() external {
  ......

    address _stabilityPool = stabilityPool;
    // deposit rewards to stability pool
    _approve(wstETH, _stabilityPool, _stabilityPoolRewards);
    IRebalancePool(_stabilityPool).depositReward(wstETH,
_stabilityPoolRewards);
  }

  if (_totalRewards > 0) {
    IERC20Upgradeable(stETH).safeTransfer(platform,
_totalRewards);
  }
}
```

Status

The team has confirmed the issue. The parameter `platform` will be set immediately after contract deployment, and thus the risk will be automatically eliminated.

4.3.3 The derivation of parameter `d[i]` in the code comments is incorrect.

Risk Type	Risk Level	Impact	Status
Comment Revise	Info	Wrong Comment	Fixed

Description

The parameter `d[i]` represents the remaining amount of assets (in fETH) for an individual user at a given moment `i`. The deduction process is as follows:

$$d_i = d_0 \prod_{n=1}^{i}(1 - \frac{L_n}{D_{n-1}})$$

$$P_i = \prod_{n=1}^{i}(1 - \frac{L_n}{D_{n-1}}) \tag{1}$$

$$Thus:$$

$$d_i = d_0 \times P_i$$

The following formula `d[i] = d[0] * P[i]` was mistakenly written as `d[i] = d[0] * P[i-1]`.

Additionally, when explaining the meaning of parameter `D[i]`, "D[i] is the total amount of assets at a[i]" should be corrected to "D[i] is the total amount of assets at time t[i]."

```
/// @dev There are 4 events:
/// + deposit some asset at t[i]
/// + withdraw some asset at t[i]
/// + liquidate at t[i]
/// + distribute reward token at t[i]
///
/// The amount of asset left after n liquidation is:
///   + d[n] = d[0] * (1 - L[1]/D[0]) * (1 - L[2]/D[1]) * ... *
(1 - L[n]/D[n-1])
///   + D[i] = D[i-1] - L[i]
/// where
///   + d[0] is the initial deposited amount of a user
///   + L[i] is the amount of liquidated asset at t[i]

      //@audit D[i] is the total amount of asset at t[i]
///   + D[i] is the total amount of asset at a[i]
/// So we need to maintain the following variables:
///   + D[i] = D[i-1] - L[i]
///   + P[i] = P[i-1] * (1 - L[i]/D[i-1])
///
/// The amount of reward token gained after n distribution is:
///   + gain = d[0] * E[1]/D[0] + d[1] * E[2]/D[1] + ... + d[n-
1]*E[n]/D[n-1]

      //@audit ... = d[0] * P[i]
///   + d[i] = d[i-1]*(1 - L[i]/D[i-1]) = d[0] * P[i-1]

///   + D[i] = D[i-1] - L[i]
/// So we need to maintain the flowing variables:
///   + D[i] = D[i-1] - L[i]
///   + P[i] = P[i-1] * (1 - L[i]/D[i-1])
///   + S[i] = S[i-1] * E[i]/D[i-1]*P[i-1]
```

Status

The team has confirmed this issue and has fixed it in commit c1b1c3f.

4.3.4 The parameter name `UserRewardSnapsot` has a spelling mistake.

Risk Type	Risk Level	Impact	Status
Code Revise	Info	Wrong Spelling	Fixed

Description

The correct spelling for the struct name should be `UserRewardSnapshot` to make the meaning more clear and coherent.

```
// @audit revise UserRewardSanpsot
struct UserRewardSnapsot {
  // The amount of pending extraRewards.
  uint256 pending;
  // The accumulated reward sum.
  uint256 accRewardsPerStake;
}

struct UserSnapshot {
  // The initial amount of asset deposited.
  uint256 initialDeposit;
  // The unlocked/unlocking state.
  UserUnlock initialUnlock;
  // The epoch state snapshot at last interaction.
  EpochState epoch;
  // The reward snapshot of base token at last interaction.
  UserRewardSnapsot baseReward;
  // Mapping from token address to extra reward snapshot.
  mapping(address => UserRewardSnapsot) extraRewards;
}
```

Status

The team has confirmed this issue and has fixed it in commit c1b1c3f.

4.3.5 Parameter discussion for distributing extra rewards during protocol liquidation.

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

Description

When performing protocol settlement, it is necessary to execute the `_distributeRewards()` function to distribute rewards. However, the code is passing the parameter `address(this)` which means updating rewards for the contract's address. In theory, the contract's address should not hold any funds, and even if a user accidentally sends funds to the contract's address while depositing, the contract itself will not be able to claim corresponding profits. From the perspective of code design logic, `address(this)` here should be modified to `address(0)`.

```
/// @inheritdoc IRebalancePool
function liquidate(uint256 _maxAmount, uint256 _minBaseOut)
  external
  override
  returns (uint256 _liquidated, uint256 _baseOut)
{
  require(liquidator == msg.sender, "only liquidator");

  // distribute pending extraRewards
  _distributeRewards(address(this));

  ITreasury _treasury = ITreasury(treasury);

  require(_treasury.collateralRatio() <
liquidatableCollateralRatio, "cannot liquidate");
    ......
}


function _distributeRewards(address _account) internal {
    uint256 length = rewards.length;
    for (uint256 i = 0; i < length; i++) {
      _distributeReward(rewards[i]);
    }

    if (_account != address(0)) {
      _updateAccountRewards(_account);
    }
 }
```

Status

The team has confirmed this issue and has fixed it in commit c1b1c3f.

4.3.6 Discussion on the user fund withdrawal plan.

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

Description

In the current implementation, users cannot initiate multiple principal withdrawal requests simultaneously. When a user initiates a principal withdrawal request, the funds are locked for an `unlockDuration` period. If the user initiates a new principal withdrawal request before the lock period has expired, the unlock duration will be recalculated. Consequently, the unlocking of the first deposit will be delayed, eventually coinciding with the unlocking of the second deposit. In the worst-case scenario, if a user initiates multiple requests to unlock funds during the lockout period, the fund unlocking will be continually delayed, resulting in a significant delay.

This risk is within the user's control since it can only be initiated by the user, and no unlocking is allowed on their behalf. It is crucial to verify that the fund retrieval process aligns with expectations and requirements.

```
/// @inheritdoc IRebalancePool
function unlock(uint256 _amount) external override {
  ......

  // @note the snapshot is updated in `_distributeRewards` once,
the value of `unlock.amount` is correct.
  UserUnlock memory _unlock =
snapshots[msg.sender].initialUnlock;

  // @audit allow users to withdraw funds again before the
previous unlock period expires.
  require(_unlock.amount == 0 || _unlock.unlockAt >
block.timestamp, "nonzero unlocked token");

  _unlock.amount = uint128(_amount.add(_unlock.amount));
  emit UserUnlockChange(msg.sender, _unlock.amount, 0);
  // @audit update the user's fund withdrawal time.
  uint256 _unlockAt = block.timestamp + unlockDuration;


  if (_unlockAt < _unlock.unlockAt) {
    _unlockAt = _unlock.unlockAt;
  }
  snapshots[msg.sender].initialUnlock.unlockAt =
uint128(_unlockAt);

  _takeAccountSnapshot(msg.sender, _newDeposit, _unlock.amount);

  totalSupply = totalSupply.sub(_amount);
  totalUnlocking = totalUnlocking.add(_amount);

  emit Unlock(msg.sender, _amount, _unlockAt);
}
```

Status

The team has confirmed the design logic.

4.3.7 There is no capability to remove potentially expired reward tokens.

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

Description

Considering the completeness of the contract's functionality, it is advisable to add a function to remove potential expired additional reward tokens, especially since the administrator can add additional reward tokens using the `addReward()` function. By incorporating this removal functionality, the contract will have improved management capabilities, ensuring a more robust and comprehensive reward token management system.

```
/// @notice Add a new reward token to this contract.
/// @param _token The address of the reward token.
/// @param _manager The address of reward token manager.
/// @param _periodLength The length of distribution period.
function addReward(
  address _token,
  address _manager,
  uint32 _periodLength
) external onlyOwner {
  require(rewardManager[_token] == address(0), "duplicated reward
token");
  require(_manager != address(0), "zero manager address");
  require(_periodLength > 0, "zero period length");

  rewardManager[_token] = _manager;
  extraRewardState[_token].periodLength = _periodLength;
  extraRewards.push(_token);

  emit AddRewardToken(_token, _manager, _periodLength);
}
```

Status

The team has confirmed this issue and has fixed it in commit c1b1c3f.

4.3.8 Discussion on the meaning of the parameter `_assetLossPerUnitStaked`.

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

Description

To calculate the remaining fETH principal and its corresponding stETH earnings for a user, you need to use the intermediate parameter P, which is calculated using the formula:

$$P_i = \prod_{n=1}^{i} \left(1 - \frac{L_n}{D_{n-1}}\right) \tag{2}$$

Where:

- P_i represents the intermediate parameter for user i.
- i is the user index or number.
- Ln is the amount of fETH principal withdrawn by user i.
- D_{n-1} is the total fETH principal deposited in the contract before user i initiated the withdrawal.

In the design logic of the f(x) protocol, users need to wait for a 2-week locking period to end before they can withdraw their funds. During the unlock period, there are no earnings, but the user's funds will still participate in the protocol's clearing process. In this situation, the funds of a user during the unlock period should also be considered in the calculation of the intermediate parameter `P`. The parameter `_assetLossPerUnitStaked` is indeed part of the formula $\frac{L_i}{D_{i-1}}$ as above mentioned. The numerator `_lossNumerator` should represent the amount of fETH that the user clears during the unlock period, while the denominator should be the total fETH token amount deposited by all users in the contract, which includes both locked and unlocked amounts, represented by `_totalSupply + totalUnlocking`.

```
function _notifyLoss(uint256 _loss) internal {
    uint256 _totalSupply = totalSupply;
    uint256 _totalUnlocking = totalUnlocking;
    uint256 _assetLossPerUnitStaked;

    if (_loss == _totalSupply + _totalUnlocking) {
      _assetLossPerUnitStaked = PRECISION;
      lastAssetLossError = 0;
      totalSupply = 0;
      totalUnlocking = 0;
    } else {
      uint256 _lossFromDeposit =
_loss.mul(_totalSupply).div(_totalUnlocking + _totalSupply);
```

```
      uint256 _lossNumerator =
_lossFromDeposit.mul(PRECISION).sub(lastAssetLossError);
      // Add 1 to make error in quotient positive. We want
"slightly too much" LUSD loss,
      // which ensures the error in any given
compoundedAssetDeposit favors the Stability Pool.

      //@audit get the value of _assetLossPerUnitStaked
      _assetLossPerUnitStaked =
(_lossNumerator.div(_totalSupply)).add(1);
      lastAssetLossError =
(_assetLossPerUnitStaked.mul(_totalSupply)).sub(_lossNumerator);

      totalSupply = _totalSupply.sub(_lossFromDeposit);
      totalUnlocking =
_totalUnlocking.sub(_loss.sub(_lossFromDeposit));
    }

    // The newProductFactor is the factor by which to change all
deposits, due to the depletion of Stability Pool LUSD in the
liquidation.
    // We make the product factor 0 if there was a pool-emptying.
Otherwise, it is (1 - LUSDLossPerUnitStaked)
    //@audit calculate the latest factor for P
    uint256 _newProductFactor =
PRECISION.sub(_assetLossPerUnitStaked);

    EpochState memory _currentEpoch = epochState;

    // If the Stability Pool was emptied, increment the epoch,
and reset the scale and product P
    if (_newProductFactor == 0) {
      _currentEpoch.epoch += 1;
      _currentEpoch.scale = 0;
      _currentEpoch.prod = uint128(PRECISION);

      // If multiplying P by a non-zero product factor would
reduce P below the scale boundary, increment the scale
    } else if
(_newProductFactor.mul(_currentEpoch.prod).div(PRECISION) <
SCALE_FACTOR) {
      _currentEpoch.prod =
uint128(_newProductFactor.mul(_currentEpoch.prod).mul(SCALE_FACTO
R).div(PRECISION));
      _currentEpoch.scale += 1;
    } else {
      _currentEpoch.prod =
uint128(_newProductFactor.mul(_currentEpoch.prod).div(PRECISION))
;
    }

    epochState = _currentEpoch;
  }
```

Suggestion

Recommended modification to the code:

```
function _notifyLoss(uint256 _loss) internal {
    uint256 _totalSupply = totalSupply;
    uint256 _totalUnlocking = totalUnlocking;
    uint256 _assetLossPerUnitStaked;

    if (_loss == _totalSupply + _totalUnlocking) {
      _assetLossPerUnitStaked = PRECISION;
      lastAssetLossError = 0;
      totalSupply = 0;
      totalUnlocking = 0;
    } else {
      uint256 _lossFromDeposit =
_loss.mul(_totalSupply).div(_totalUnlocking + _totalSupply);

      // @audit modify the following code
      uint256 _lossNumerator =
_loss.mul(PRECISION).sub(lastAssetLossError);
      // Add 1 to make error in quotient positive. We want
"slightly too much" LUSD loss,
      // which ensures the error in any given
compoundedAssetDeposit favors the Stability Pool.

      // @audit modify the following code
      _assetLossPerUnitStaked = (_lossNumerator.div(_totalSupply
+ _totalUnlocking)).add(1);
      lastAssetLossError =
(_assetLossPerUnitStaked.mul(_totalUnlocking +
_totalSupply)).sub(_lossNumerator);


      totalSupply = _totalSupply.sub(_lossFromDeposit);
      totalUnlocking =
_totalUnlocking.sub(_loss.sub(_lossFromDeposit));
    }

    // The newProductFactor is the factor by which to change all
deposits, due to the depletion of Stability Pool LUSD in the
liquidation.
    // We make the product factor 0 if there was a pool-emptying.
Otherwise, it is (1 - LUSDLossPerUnitStaked)

    uint256 _newProductFactor =
PRECISION.sub(_assetLossPerUnitStaked);

    EpochState memory _currentEpoch = epochState;

    ......
  }
```

Status

The team has fixed this issue in commit c1b1c3f.

4.3.9 Add checks for the parameters `_initialDeposit` and `_initialUnlock` to save gas.

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

Description

The parameters `_initialDeposit` and `_initialUnlock` represent the user's deposited principal amount and the amount of funds awaiting unlocking in the contract. Both of these parameters may be 0. For instance, when a user withdraws all their deposited funds from the protocol, their corresponding `initialDeposit` will be 0. Similarly, when a user has no funds awaiting unlocking, the parameter `_initialUnlock` will be 0. In both cases where these parameters are 0, the clearing action will not affect the corresponding share (which remains constant at 0). Therefore, there is no need to execute `_getCompoundedStakeFromSnapshots()` to calculate the compounded share after clearing, as this would waste gas.

```
/// @dev Internal function to update account reward accumulation.
/// @param _account The address of user to update.
function _updateAccountRewards(address _account) internal {
  uint256 _initialDeposit = snapshots[_account].initialDeposit;
  uint256 _initialUnlock =
snapshots[_account].initialUnlock.amount;

  ......

  // 3. update possible asset loss from the deposited assets
  // @audit when _initialDeposit is 0, there is no need to
calculate the _compoundedDeposit value
  uint256 _compoundedDeposit =
_getCompoundedStakeFromSnapshots(_initialDeposit,
snapshots[_account].epoch);
  if (_compoundedDeposit != _initialDeposit) {
    emit UserDepositChange(_account, _compoundedDeposit,
_initialDeposit.sub(_compoundedDeposit));
    snapshots[_account].initialDeposit = _compoundedDeposit;
  }

  // 4. update possible asset loss from the unlocking assets
  // @audit when _initialUnlock is 0, there is no need to
calculate the _compoundedUnlock value
  uint256 _compoundedUnlock =
_getCompoundedStakeFromSnapshots(_initialUnlock,
snapshots[_account].epoch);
```

```
  if (_compoundedUnlock != _initialUnlock) {
    emit UserUnlockChange(_account, _compoundedUnlock,
_initialUnlock.sub(_compoundedUnlock));
    snapshots[_account].initialUnlock.amount =
uint128(_compoundedUnlock);
  }

  snapshots[_account].epoch = _currentEpoch;
}
```

Status

The team has confirmed this issue and has fixed it in commit c1b1c3f.

5. Conclusion

After auditing and analyzing the RebalancePool contract, 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