Security Audit Report

f(x) Protocol 2.1 by AladdinDAO

SECBIT

Aug 13, 2025

1. Introduction

The AladdinDAO is a decentralized network to shift crypto investments from venture capitalists to the wisdom of crowds through collective value discovery. The f(x) Protocol is a decentralized system built to deliver high-leverage trading and yield-bearing stablecoins without relying on centralized intermediaries. Version 2.0 introduced xPOSITIONs, enabling fully collateralized long exposure with leverage up to 7x. These positions are notable for their minimal liquidation risk, absence of recurring borrowing costs, and on-chain modularity, providing users with real leverage rather than synthetic exposure. Version 2.1 expands the protocol's functionality by introducing sPOSITIONs, which enable fixed-leverage short exposure. Unlike xPOSITIONs, sPOSITIONs are collateralized using fxUSD and are executed atomically, eliminating the need for active collateral management or monitoring. sPOSITIONs feature no margin calls, no borrowing interest, and maintain the same liquidation brake mechanism that helped make xPOSITIONs robust and reliable. SECBIT Labs conducted an audit from May 13, 2025, to July 22, 2025, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that this f(x) Protocol 2.1 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).

Update on August 13, 2025:

The f(x) protocol has implemented a new protective upgrade to address potential risks and enhance protocol security. We have reviewed the changes in commit *068ab89* and assessed their impact on the protocol's security.

The principle behind this change is the implementation of a transaction-level lock using Transient Storage (EIP-1153). This lock is automatically released upon completion of the transaction. The purpose is to protect certain critical functions by ensuring they can only be called once within the same transaction, thereby guaranteeing that contract function calls adhere to their original design. A side effect is that any attempt to combine calls to these functions multiple times within a single transaction will cause the transaction to fail.

We have evaluated this change and consider it effective. It will not affect ordinary users, and we have not identified any new security issues.

Type	Description	Level	Status
Parameter Naming Style	4.3.1 It is recommended to rename input parameters in `redeemByCreditNote()`, `rebalance()`, and `liquidate()` to reflect their actual use and improve future maintainability.	Info	Fixed
Design & Implementation	4.3.2 It is recommended to call `_updateCollAndDebtIndex()` to update the `collIndex` parameter before checking whether the short pool is insolvent.	Low	Fixed
Design & Implementation	4.3.3 Discussion on short pool insolvency handling logic.	Info	Discussed
Design & Implementation	4.3.4 There is an error in the calculation of the parameter `bonusFromReserve`.	High	Fixed
Design & Implementation	4.3.5 Incorrect calculation of the amount of debt to be repaid by the administrator.	High	Fixed
Design & Implementation	4.3.6 Discussion of the `referral` parameter.	Info	Fixed
Design & Implementation	4.3.7 Incorrect protocol revenue calculation logic in long pool.	High	Fixed
Design & Implementation	4.3.8 The long pool uses an incorrect function, `_scaleUp()`, to calculate the amount of actual collateral to be deducted when covering bad debt from the short pool.	High	Fixed
Design & Implementation	4.3.9 Discussion of the logic of the `_transferCollateralOut()` function.	High	Fixed
Gas Optimization	4.3.10 It is recommended to remove the conditional check for the `_checkValueTooLarge()` function within the `_updateMiscData()` function to save gas.	Info	Fixed
Design & Implementation	4.3.11 Logic discussion for the function `onCloseOrRemoveShortPositionFlashLoan()`.	Info	Discussed
Design & Implementation	4.3.12 Adjust the decimals of the `CreditNote` token to suit the actual scenario.	Low	Fixed
Design & Implementation	4.3.13 Discussion on the fee mechanism for redeeming fxUSD tokens via the `redeemByCreditNote()` function in the `ShortPoolManager` contract.	Info	Discussed
Design & Implementation	4.3.14 High-leverage user positions may be reduced as a result of collateral redemption actions.	Info	Discussed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the f(x) Protocol 2.1 is shown below:

- Smart contract code
    - f(x) Protocol 2.1
        - initial review commit *4c16bea*
        - final review commit *23db2a8*
    - `FeeDistributorAdmin` contract
        - review commit *6164e28*

2.2 Contract List

The following content shows the contracts included in the f(x) Protocol 2.1, which the SECBIT team audits:

Name	Lines	Description
AaveFundingPool.sol	60	The supplementary module of the protocol updates the collateral index system, which is utilized for calculating and collecting funding costs.
BasePool.sol	480	The core module of the protocol executes critical functionalities, including position opening, position closing, position adjustment, and position liquidation, implementing the protocol's fundamental operational mechanisms.
PoolErrors.sol	30	The auxiliary component of the protocol documents the enumerated error states and their associated failure conditions encountered during protocol operations.
PoolStorage.sol	212	The auxiliary component of the protocol documents essential protocol metrics, encompassing debt liquidation boundaries, protocol-wide collateral aggregation, and cumulative debt quantification.
PositionLogic.sol	88	The supplementary module of the protocol calculates user position metrics, including collateral quantities and debt obligations, as well as residual capital following position liquidation procedures.
TickLogic.sol	203	The supplementary module of the protocol calculates corresponding tick values based on position debt ratios, facilitating position liquidation processes through tick-based execution mechanisms.
CreditNote.sol	26	The debt tokens allow their holder to redeem fxUSD tokens from the short pool.

ShortPool.sol	70	It inherits all functionalities of the `BasePool` contract and additionally introduces an interface for redeeming debt tokens within the `ShortPool` contract.
ShortPoolManager.sol	472	The peripheral contract provides direct interfaces for users to execute short position operations, including position opening, closing, adjustment, and liquidation procedures.
FlashLoans.sol	42	The supplementary module of the protocol implements flash loan functionality, enabling users to access protocol-controlled capital through instantaneous borrowing mechanisms.
FxUSDPriceOracle.sol	62	Price oracle contracts that retrieve the secondary market price of the fxUSD token.
PegKeeper.sol	123	The supplementary module of the protocol maintains stability in the exchange ratio between fxUSD tokens and USDC tokens.
PoolConfiguration.sol	261	Fee configuration contracts that allow the protocol to set and manage various fee parameters.
PoolManager.sol	699	The peripheral contract provides direct interfaces for users to execute position operations, including long position opening, closing, adjustment, and liquidation procedures.
ProtocolFees.sol	215	The contract configures protocol fee parameters, including position opening fees, closing fees, liquidation fees, and other associated cost metrics.
FlashLoanCallbackFacet.sol	36	The flash loan interface function of the Balancer v2 contract facilitates capital return following the execution of protocol-specific operational logic.
FlashLoanFacetBase.sol	37	A base contract serving flash loan functionality.
FxUSDBasePoolFacet.sol	64	The peripheral contract facilitates legacy fx protocol users in directly depositing their fTokens into the `fxBASE` contract to receive corresponding share allocations.
LongPositionEmergencyCloseFacet.sol	210	Flash close contracts for long pools, enabling users to repay and close leveraged long positions using flash loans in either fxUSD tokens or USDC tokens.
ShortPositionOperateFlashLoanFacet.sol	182	Flash open and close contracts for short pools, allowing users to both open and close short positions atomically through flash loans.
InverseWstETHPriceOracle.sol	33	Oracle integration for short pools, which provides real-time fxUSD-LSD (Liquid Staking Derivative) price feeds.
FeeDistributorAdmin.sol	26	The contract allows the designated keeper to trigger the distribution of accumulated protocol fees held in the `FeeDistributor` contract at any time.

3. Contract Analysis

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

3.1 Role Classification

Two key roles in the f(x) Protocol 2.1 are Governance Account and Common Account.

- Governance Account
    - Description

    Contract Administrator, including default administrator role, emergency role, debt reducer role, harvester role, pool killer role, and multi-call role.

  - Authority
    - Update the fees ratio
    - Update market and incentive configurations
    - Transfer ownership
    - Pause crucial functions
    - Claim protocol revenue and distribute it
  - 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 2.1
  - Authority
    - Open and close positions using authorized tokens, resulting in fixed leverage allocation
    - Redeem base tokens with fxUSD tokens by long pool
    - Rebalance user positions to prevent excessive debt ratios
    - Liquidate position and manage protocol bad debt resolution
  - Method of Authorization

    No authorization required

3.2 Functional Analysis

f(x) Protocol v2.1 builds upon the foundational mechanisms established in v1.0, retaining the robust, yield-bearing stablecoin design of the fxUSD token while significantly enhancing its leveraged trading infrastructure. This upgrade refines the xPOSITIONs architecture for leveraged long exposure and introduces sPOSITIONs, enabling fixed-leverage short exposure — both with minimal liquidation risk, no recurring funding costs, and atomic, trustless execution. 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 three parts:

PoolManager & ShortPoolManager

These contracts provide core protocol interfaces for users, encompassing position opening, closing, adjustment, liquidation operations, and protocol funding cost collection mechanisms. The main functions in these contracts are as follows:

- `operate()`

  The function enables users to initiate long or short positions and execute position adjustments.

- `redeem()` (Only supports asset redemption from the long pool)

  Users holding fxUSD tokens obligations can invoke this function to redeem corresponding collateral assets.

- `rebalance()`

  Any user can invoke this function to redeem position debt, reduce leverage, and mitigate system debt risk when collateral prices decline, resulting in increased debt ratios.

- `liquidate()`

  When position debt ratios exceed liquidation thresholds, users can invoke this function to execute position liquidation and receive liquidation incentives.

- `harvest()`

  The protocol assesses time-based funding costs on user-deposited collateral, enabling the harvester to distribute rewards through this function while receiving a proportional share of distribution incentives.

LongPositionEmergencyCloseFacet

In f(x) Protocol v2.1, the introduction of short positions allows a portion of the collateral from the long pool to be temporarily borrowed to support short-side leverage. When the available collateral in the long pool becomes insufficient, users redeeming their positions may receive CreditNote tokens in lieu of the full collateral amount. To address this scenario, a dedicated contract enables users to redeem CreditNote tokens for fxUSD token, and subsequently convert the received fxUSD token into a user-specified token, ensuring a seamless and user-friendly redemption process even under constrained collateral conditions. The main functions in this contract are as follows:

- `closeOrRemoveLongPositionFlashLoan()`

  Any user can invoke this function to close a position or remove collateral from a position.

- `closeOrRemoveLongPositionFlashLoanWithUSDC()`

Any user can invoke this function to close a position or remove collateral from a position with USDC tokens.

ShortPositionOperateFlashLoanFacet

This contract assists users in position opening and closing operations through flash loan mechanisms, enabling full sPOSITION position allocation. The main functions in this contract are as follows:

- `openOrAddShortPositionFlashLoan()`

  Users can provide any tokens, and this function assists in converting them into specified collateral for position opening or expansion. Users seeking full sPOSITION allocation can utilize the function's integrated flash loan capabilities simultaneously.
- `closeOrRemoveShortPositionFlashLoan()`

  Users can utilize flash loan functionality to close or partially reduce positions.

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 for 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 It is recommended to rename input parameters in `redeemByCreditNote()`, `rebalance()`, and `liquidate()` to reflect their actual use and improve future maintainability.

Risk Type	Risk Level	Impact	Status
Parameter Naming Style	Info	Confusing Parameter Naming	Fixed

Location

ShortPoolManager.sol#L325

ShortPoolManager.sol#L358

ShortPoolManager.sol#L391

ShortPoolManager.sol#L409

ShortPoolManager.sol#L427

Description

Based on the contract logic and the protocol's established naming conventions within the short pool module, the input parameter `debts` typically refers to the collateral amount denominated in wstETH token, while `rawDebts` refers to the actual protocol debt denominated in stETH token. These two values are interconverted via the `scalingFactor` parameter.

Currently, only the `redeem()` function in the `ShortPoolManager` contract adheres to this convention, correctly using `debts` to represent wstETH token amounts. In contrast, the `redeemByCreditNote()`, `rebalance()`, and `liquidate()` functions use parameter names (`debts`, `maxDebt`) to represent stETH-denominated debt, which deviates from the established naming standard. This inconsistency may lead to confusion or misinterpretation during code maintenance and further development.

Furthermore, the `_checkDebtValues()` function is designed to prevent redemptions of extremely small debt amounts, which could otherwise be exploited by malicious actors to initiate a high number of minimal redemptions. This could lead to long and gas-expensive internal redemption chains, potentially obstructing the closure of highly leveraged positions due to block gas limits.

Currently, `_checkDebtValues()` is used to enforce minimum redemption thresholds in both `redeem()` (for wstETH-denominated debts) and `redeemByCreditNote()` (for stETH-denominated debts). This mixed usage may cause ambiguity and weaken the semantic clarity of the debt checks.

```
function redeem(
  address pool,
  uint256 debts,  // @audit this parameter represents the number of
wstETH
  uint256 minColls
) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls) {

    // @audit the wstETH amount must not be less than `debts`
    _checkDebtValues(pool, debts);

    address debtToken = IShortPool(pool).debtToken();
    IERC20(debtToken).safeTransferFrom(_msgSender(), address(this),
debts);

    uint256 scalingFactor = _getTokenScalingFactor(debtToken);
    uint256 rawDebts = _scaleUp(debts, scalingFactor);
    ......
  }

 function redeemByCreditNote(
    address pool,
    uint256 debts, // @audit this parameter represents the number of
stETH
    uint256 minColls
  ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls) {

    // @audit the stETH amount must not be less than `debts`
    _checkDebtValues(pool, debts);

    address creditNote = IShortPool(pool).creditNote();
    IERC20(creditNote).safeTransferFrom(_msgSender(), address(this),
debts);

    if (debts > IShortPool(pool).getTotalRawDebts()) {
      revert ErrorRedeemExceedTotalDebt();
    }

   ......
  }

function rebalance(
    address pool,
    address receiver,
    int16 tick,
    uint256 maxDebt // @audit this parameter represents the number of
stETH
  ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls, uint256 debts) {

  // @audit the stETH amount must not be less than `maxDebt`
    _checkDebtValues(pool, maxDebt);

    LiquidateOrRebalanceMemoryVar memory op =
_beforeRebalanceOrLiquidate(pool);
    IShortPool.RebalanceResult memory result =
IShortPool(pool).rebalance(tick, maxDebt);
  ......
 }

 function rebalance(
    address pool,
    address receiver,
    uint256 maxDebt // @audit this parameter represents the number of
stETH
  ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls, uint256 debts) {

    // @audit the stETH amount must not be less than `maxDebt`
    _checkDebtValues(pool, maxDebt);


     LiquidateOrRebalanceMemoryVar memory op =
_beforeRebalanceOrLiquidate(pool);
    IShortPool.RebalanceResult memory result =
IShortPool(pool).rebalance(maxDebt);
    ......
  }


function liquidate(
    address pool,
    address receiver,
    uint256 maxDebt // @audit this parameter represents the number of
stETH
  ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls, uint256 debts) {

    // @audit the stETH amount must not be less than `maxDebt`
    _checkDebtValues(pool, maxDebt);

    (bool underCollateral, uint256 shortfall) =
IShortPool(pool).isUnderCollateral();

    ......

    LiquidateOrRebalanceMemoryVar memory op =
_beforeRebalanceOrLiquidate(pool);
    {
      IShortPool.LiquidateResult memory result;
      uint256 reservedColls =
IReservePool(reservePool).getBalance(fxUSD);

      result = IShortPool(pool).liquidate(maxDebt, reservedColls);
      op.rawColls = result.rawColls + result.bonusRawColls;
      op.bonusRawColls = result.bonusRawColls;
      op.rawDebts = result.rawDebts;

      // take bonus or shortfall from reserve pool
      uint256 bonusFromReserve = result.bonusFromReserve;
      if (bonusFromReserve > 0) {
        bonusFromReserve = _scaleDown(result.bonusFromReserve,
op.scalingFactor);
        IReservePool(reservePool).requestBonus(fxUSD, address(this),
bonusFromReserve);

        // increase pool reserve first
        _changePoolCollateral(pool, int256(bonusFromReserve));
      }
    }

    (colls, debts) = _afterRebalanceOrLiquidate(pool, op, receiver);

    emit Liquidate(pool, colls, debts);
  }
```

Suggestion

It was recommended to follow the original naming convention by renaming the `debts` parameter in the `redeemByCreditNote()` function to `rawDebts`, and the `maxDebt` parameters in both the `rebalance()` and `liquidate()` functions to `maxRawDebt`, to improve code clarity and maintainability.

Status

The development team has adopted this suggestion and updated the parameter names accordingly in commit d09d2ad, aligning them with the protocol's naming conventions.

4.3.2 It is recommended to call `_updateCollAndDebtIndex()` to update the `collIndex` parameter before checking whether the short pool is insolvent.

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

Location

ShortPoolManager.sol#L431

Description

In the short pool, any user can call the `liquidate()` function to liquidate positions when either a highly leveraged position reaches the liquidation threshold or the total collateral value in fxsud falls below the total debt, resulting in insolvency at the pool level. The `liquidate()` function first calls the `isUnderCollateral()` function in the short pool contract to determine whether the pool is undercollateralized.

It was observed that `liquidate()` does not invoke the _updateCollAndDebtIndex() function prior to this check. As a result, the `collIndex` parameter may not be up to date, potentially excluding accrued but uncollected funding costs from the collateral calculation.

Under the current implementation, this could lead to inaccuracies in determining the pool's collateralization status. Further analysis of the `isUnderCollateral()` function reveals that without updating `collIndex`, the computed `totalRawColls` may be overstated, which can result in two possible outcomes:

(1). If the pool is already undercollateralized (`underCollateral == true`), the overstated `totalRawColls` would cause the calculated system shortfall to be underestimated.

(2). Suppose the actual collateral amount after deducting the accrued funding cost is denoted as `totalRawColls'`, then:

$$\text{totalRawColls} > \text{totalRawColls'} \tag{1}$$

This could result in a scenario where:

$$\text{totalRawColls} \times \text{price} > \text{totalRawDebts} \times \text{PRECISION} > \text{totalRawColls'} \times \text{price} \quad (2)$$

That is, the system has already incurred a shortfall, but the uncollected funding cost causes the `liquidate()` function to fail to recognize the undercollateralization, delaying liquidation and bad debt resolution.

```
function liquidate(
  address pool,
  address receiver,
  uint256 maxDebt
) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls, uint256 debts) {
  _checkDebtValues(pool, maxDebt);

   // @audit the internal function did not update the funding cost in
advance
  (bool underCollateral, uint256 shortfall) =
IShortPool(pool).isUnderCollateral();
  if (underCollateral) {
    ......
    if (rawReserveFund >= shortfall) {
      rawReserveFund = shortfall;
      shortfall = 0;
    } else {
      shortfall -= rawReserveFund;
    }
   ......
}
```

```
// @audit ShortPool.sol#L71-L80
function isUnderCollateral() external view returns (bool underCollateral,
uint256 shortfall) {
    // @audit read the total amount of debt (stETH token) and collateral
(fxUSD token) for all users in the short pool
    (uint256 totalDebtShares, uint256 totalColls) =
_getDebtAndCollateralShares();

    // @audit fxUSD token price relative to stETH token
    uint256 price = IPriceOracle(priceOracle).getLiquidatePrice();

    // @audit read the current `debtIndex` and `collIndex`, but these may
not be the latest values.
```

```
    (uint256 debtIndex, uint256 collIndex) =
_getDebtAndCollateralIndex();

    // @audit after considering the funding cost charged by the protocol,
if the `collIndex` of the collateral fxUSD token held by all users is not
updated in a timely manner, it may result in an inflated `totalRawColls`
value.
    uint256 totalRawColls = _convertToRawColl(totalColls, collIndex,
Math.Rounding.Down);

    // @audit after considering bad debts in the system (which may also
be repaid by administrators on behalf of users), the total amount of debt
owed by all users.
    uint256 totalRawDebts = _convertToRawDebt(totalDebtShares, debtIndex,
Math.Rounding.Down);
    // @audit determine whether the short pool is undercollateralized
    underCollateral = totalRawDebts * PRECISION >= totalRawColls * price;

    // @audit calculate the number of bad debts
    shortfall = underCollateral ? totalRawDebts - (totalRawColls * price)
/ PRECISION : 0;
  }
```

Status

The development team has confirmed this issue and modified the logic of the `liquidate()` function in commit d09d2ad . The new code adjusts the internal logic of the `isUnderCollateral()` function, which calls the `_updateCollAndDebtIndex()` function within it to update the `collIndex` parameter, thereby deducting the funding cost funds to be charged from the user's collateral funds in advance.

4.3.3 Discussion on short pool insolvency handling logic.

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

Location

ShortPoolManager.sol#L453-L460

Description

Suppose the short pool becomes undercollateralized (i.e., the total collateral value in fxUSD tokens falls below the total debt). In that case, any user may call the `liquidate()` function to repay all outstanding debt on behalf of the pool. In this case, the liquidator is not required to cover the bad debt portion and will receive all fxUSD tokens collateral held by the short pool. Following liquidation, the protocol invokes the `kill()` operation, disabling all core functionalities for short pool users, including position opening/closing, rebalancing, and redemption.

From the user's perspective, undercollateralization at the pool level typically results from delayed rebalancing or liquidation of highly leveraged positions. Under the current design, however, the resulting bad debt is effectively socialized and absorbed by users with lower leverage, which may be perceived as unfair and could negatively impact user experience.

The following scenario illustrates this issue:

Assume the price of the stETH token is $3,000 (for simplicity, conversions from wstETH token to stETH token are ignored), and the price of fxUSD token is fixed at $1. Two users, userA and userB, open short positions at 10x and 5x leverage, respectively.

(1).UserA (10x leverage) deposits fxUSD token worth $1,000,000:

$$\begin{gathered} leverage = \frac{n_f}{sPostion} = \frac{100 \times 10^4}{sPostion} = 10 \\ sPostion = 10 \times 10^4 \end{gathered} \tag{3}$$

Thus, userA borrows 300 stETH token:

$$\begin{gathered} n_f = n^- s + sPostion \\ n^- = \frac{n_f - sPostion}{s} = \frac{100 \times 10^4 - 10 \times 10^4}{3000} = 300 \end{gathered} \tag{4}$$

(2). UserB (5x leverage) deposits fxUSD token worth $600,000:

$$\begin{gathered} leverage = \frac{n_f}{sPostion} = \frac{60 \times 10^4}{sPostion} = 5 \\ sPostion = 12 \times 10^4 \end{gathered} \tag{5}$$

Thus, userB borrows 160 stETH token:

$$n_f = n^- s + sPostion$$

$$n^- = \frac{n_f - sPostion}{s} = \frac{60 \times 10^4 - 12 \times 10^4}{3000} = 160 \tag{6}$$

(3). Suppose the price of stETH token increases to $3,500. The debt value of userA's 300 stETH token is: $300 \times 3500 = 1,050,000 \text{ USD}$, which exceeds userA's collateral value ($1,000,000), making the position undercollateralized. Meanwhile, userB's 160 stETH token has a debt value of: $160 \times 3500 = 560,000 \text{ USD}$, which is still below userB's collateral value ($600,000). The total collateral in the pool is $1,600,000, while the total debt is: $1,050,000 + 560,000 = 1,610,000 \text{ USD}$, indicating that the short pool is now collectively undercollateralized.

(4). Any user can now call `liquidate()` to process the pool-level insolvency. As a result:

UserA, whose position triggered the shortfall, effectively profits by $50,000, as the value of their borrowed assets exceeds their collateral.

UserB, despite maintaining a healthy leverage ratio and not reaching liquidation thresholds, will have their position forcibly closed and will incur a $40,000 loss.

This behavior results in wealth transfer from low-leverage users (such as userB) to high-leverage users (like userA). The profit of $50,000 obtained by userA is ultimately sourced from the excess collateral of userB ($40,000), and the protocol's `reservePool` ($10,000). If the reserve is insufficient, the remaining amount may be covered via funding cost deductions from long pool users' collateral.

```
function liquidate(
    address pool,
    address receiver,
    uint256 maxDebt
  ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls, uint256 debts) {
    _checkDebtValues(pool, maxDebt);

    (bool underCollateral, uint256 shortfall) =
IShortPool(pool).isUnderCollateral();
    if (underCollateral) {
      address debtToken = IShortPool(pool).debtToken();
      uint256 scalingFactor = _getTokenScalingFactor(debtToken);
      // user must liquidate all
      colls = IShortPool(pool).getTotalRawCollaterals();
      uint256 rawDebts = IShortPool(pool).getTotalRawDebts();
      // user needs to pay (rawDebts - shortfall)
      debts = _scaleDown(rawDebts - shortfall, scalingFactor);
```

```
 // @audit the caller transfers the total amount of debt (excluding
system bad debt) to the short pool under the contract, calculated in
wstETH token.
 IERC20(IShortPool(pool).debtToken()).safeTransferFrom(_msgSender(),
address(this), debts);
      // request shortfall from reserve pool first
      uint256 rawReserveFund =
_scaleUp(IReservePool(reservePool).getBalance(debtToken), scalingFactor);
      if (rawReserveFund >= shortfall) {
        rawReserveFund = shortfall;
        shortfall = 0;
      } else {
        shortfall -= rawReserveFund;
      }
      if (rawReserveFund > 0) {
        IReservePool(reservePool).requestBonus(debtToken, address(this),
_scaleDown(rawReserveFund, scalingFactor));
      }
      // repay user funds + reserve fund
      ILongPoolManager(counterparty).repay(
        IShortPool(pool).counterparty(),
        pool,
        _scaleUp(debts, scalingFactor) + rawReserveFund,
        shortfall
      );

      // @audit the caller takes all collateral from the short pool fxUSD
token.
      _transferOut(fxUSD, colls, _msgSender());

      // @audit disable all core functions of the short pool contract,
including operating, rebalancing, and liquidation positions.
      IShortPool(pool).kill();
      return (colls, debts);
    }

   ......
  }
```

Status

The development team acknowledged this issue and modified the code logic in commit d09d2ad. Considering that the secondary market may lack sufficient liquidity to absorb the fxUSD token collateral during short pool insolvency, the team opted to utilize the redemption functionality of the f(x) protocol's long pool to process the fxUSD tokens. This approach may partially dilute the leverage of high-leverage users in the long pool. In practice, the protocol incorporates multiple risk control mechanisms to mitigate the likelihood of systemic insolvency. Specifically, within the short pool:

(1). When the price of the debt asset (wstETH token) rises and causes a position's debt ratio to reach the rebalance threshold, a protocol keeper (or any user) can promptly adjust the overleveraged position via rebalancing.

(2). If the debt ratio further increases and crosses the liquidation threshold (currently set at 95%), a keeper (or any user) can trigger liquidation, during which the liquidator receives an enhanced incentive.

Only when both the rebalance and liquidation defenses fail would the protocol enter a state of systemic undercollateralization; in this case, excess collateral from lower-leverage users may be used to cover outstanding protocol debt. As a result, the probability of full short pool insolvency is considered low under the current design.

4.3.4 There is an error in the calculation of the parameter `bonusFromReserve`.

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

Location

ShortPoolManager.sol#L476

Description

The f(x) protocol incorporates a `reserve pool` contract, which is designed to provide supplementary compensation to liquidators in cases where the collateral from a liquidated position is insufficient. In the context of the short pool, where the collateral is fxUSD token and the debt asset is wstETH token, the parameter `bonusFromReserve` represents the amount of fxUSD tokens retrieved from the reserve pool.

In such cases, there is no need to apply the `_scaleDown()` function, as the retrieved funds are already denominated in the correct collateral token. Additionally, the parameter `op.scalingFactor`, which reflects the real-time conversion rate between wstETH token and stETH token, is not applicable for computing values related to the fxUSD token.

```
function liquidate(
    address pool,
    address receiver,
    uint256 maxDebt
  ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls, uint256 debts) {
    _checkDebtValues(pool, maxDebt);

    ......

    LiquidateOrRebalanceMemoryVar memory op =
_beforeRebalanceOrLiquidate(pool);
    {
      IShortPool.LiquidateResult memory result;

      // @audit read the quantity of fxUSD token under reserve pool
contract
      uint256 reservedColls =
IReservePool(reservePool).getBalance(fxUSD);
      result = IShortPool(pool).liquidate(maxDebt, reservedColls);
      op.rawColls = result.rawColls + result.bonusRawColls;
      op.bonusRawColls = result.bonusRawColls;
      op.rawDebts = result.rawDebts;

      // take bonus or shortfall from reserve pool
      // @audit the amount of fxUSD token to be paid from the reserve
pool
      uint256 bonusFromReserve = result.bonusFromReserve;
      if (bonusFromReserve > 0) {
        // @audit `bonusFromReserve` is the fxUSD token amount, but
`scalingFactor` is the wstETH token conversion factor, which should be
incorrect
        bonusFromReserve = _scaleDown(result.bonusFromReserve,
op.scalingFactor);
        // @audit retrieve fxUSD token
        IReservePool(reservePool).requestBonus(fxUSD, address(this),
bonusFromReserve);

        // increase pool reserve first
        _changePoolCollateral(pool, int256(bonusFromReserve));
      }
    }

    (colls, debts) = _afterRebalanceOrLiquidate(pool, op, receiver);
```

```
        emit Liquidate(pool, colls, debts);
    }
```

Status

Before submitting the report, the development team had already identified the issue and fixed it in commit 37a0a17.

4.3.5 Incorrect calculation of the amount of debt to be repaid by the administrator.

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

Location

ShortPoolManager.sol#L550

Description

The protocol administrator can call the `reduceDebt()` function to repay outstanding debt on behalf of all users. This function serves two primary purposes:

(1). Systemic bad debt repayment: When the protocol incurs a system-wide shortfall, the corresponding debt is proportionally distributed across all user positions. In such cases, the administrator may use this function to repay the bad debt, thereby reducing or eliminating the additional debt burden on users.

(2). User incentive: Even when no systemic bad debt exists, the administrator can invoke this function to help reduce users' outstanding debt, effectively lowering their leverage ratios. This can be used as an incentive for existing users.

In the short pool, the actual collateral asset (debtToken) is the wstETH token. When the administrator calls `reduceDebt()`, the transferred token is the wstETH token, but the protocol's core accounting is denominated in the stETH token. Therefore, the input `amount` must be converted from wstETH token to stETH token using the _scaleUp() function.

However, it was observed that the current implementation incorrectly applies the `_scaleDown()` function to perform this conversion, which may lead to inaccuracies in debt reduction calculations.

```
  function reduceDebt(
      address pool,
      uint256 amount
    ) external onlyRole(DEBT_REDUCER_ROLE) onlyRegisteredPool(pool)
  nonReentrant {
      address debtToken = IShortPool(pool).debtToken();

      // @audit transfer wstETH token
      IERC20(debtToken).safeTransferFrom(_msgSender(), address(this),
  amount);

      uint256 scalingFactor = _getTokenScalingFactor(debtToken);

      // @audit use _scaleUp() function?
      uint256 rawAmount = _scaleDown(amount, scalingFactor);

      IShortPool(pool).reduceDebt(rawAmount);
      ILongPoolManager(counterparty).repay(IShortPool(pool).counterparty(),
  pool, rawAmount, 0);

      emit ReduceDebt(pool, amount);
    }
```

Status

Before submitting the report, the development team had already identified the issue and fixed it in commit b9924d3.

4.3.6 Discussion of the `referral` parameter.

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

Location

PoolManager.sol#L369

ShortPoolManager.sol#L265

Description

In both the long pool and short pool, users can specify a `referral` address when performing position-related operations such as opening, adjusting, or closing positions. The designated referral address is eligible to receive a reward, calculated as a percentage of the user's principal. This reward is deducted directly from the user's funds and transferred to the specified referral address.

It was observed that the `referral` parameter is user-supplied and not subject to strict validation, which may result in the following issues:

(1). Referral address set to 0: If the user sets `vars.referral` to the zero address while other referral-related parameters (such as the referral ratio) are non-zero, the operation may revert. This is because the system could attempt to perform actions such as transferring funds or minting the fxUSD token to the zero address, which are disallowed.

(2). Unrestricted referral ratio: If the user provides a non-zero `vars.referral` address, the current implementation does not enforce an upper bound on the referral ratio. A malicious referrer could exploit this by setting an excessively high referral ratio, potentially resulting in significant—or even total—loss of the user's principal during the operation.

```
function operate(
    address pool,
    uint256 positionId,
    int256 newColl,
    int256 newDebt,
    bytes32 referral, // @audit any parameters can be entered by the user
    bool useStable
  ) public onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256) {
    OperationMemoryVar memory vars;

    if (useStable) {
      if (!IPoolConfiguration(configuration).isStableRepayAllowed())
revert ErrorStableRepayNotAllowed();
      if (newColl > 0 || newDebt >= 0 || positionId == 0) revert
ErrorInvalidOperation();
      vars.stablePrice =
IFxUSDBasePool(fxBASE).getStableTokenPriceWithScale();
    }

    address collateralToken = ILongPool(pool).collateralToken();
    uint256 scalingFactor = _getTokenScalingFactor(collateralToken);

    (
      vars.poolSupplyFeeRatio,
      vars.poolWithdrawFeeRatio,
      vars.poolBorrowFeeRatio,
      vars.poolRepayFeeRatio
    ) = IPoolConfiguration(configuration).getPoolFeeRatio(pool,
_msgSender());
```

```
    // @audit when the referral address is 0 and other ratios are not 0,
  it may cause the relevant operation to revert
    (
      vars.referral,
      vars.referralSupplyFeeRatio,
      vars.referralWithdrawFeeRatio,
      vars.referralBorrowFeeRatio,
      vars.referralRepayFeeRatio
    ) = _decodeReferralData(referral);

  ......
  }
```

Status

The development team removed the referral logic in commit d09d2ad.

4.3.7 Incorrect protocol revenue calculation logic in long pool.

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

Location

PoolManager.sol#L634-L645

Description

In the long pool, protocol revenue consists of two components: funding cost and exchange rate.

Funding cost charged on user positions, calculated based on the notional value of assets denominated in stETH tokens, and exchange rate gain arising from the increasing conversion rate between the protocol's actual collateral (wstETH token) and the accounting unit used for user positions (stETH token). All user position balances are recorded in terms of stETH token, while the protocol holds wstETH token as the actual underlying asset. The relationship between the two tokens is defined as:

$$\text{wstETH amount} = \text{stETH amount} \times \text{scalingFactor} \tag{7}$$

The `scalingFactor` represents the real-time exchange rate between stETH token and wstETH token. It increases monotonically over time and can be queried via the `getRate()` function.

```
function harvest(
    address pool
  )
    external
    onlyRegisteredPool(pool)
    onlyRole(HARVESTER_ROLE)
    nonReentrant
    returns (uint256 amountRewards, uint256 amountFunding)
  {
    address collateralToken = ILongPool(pool).collateralToken();
    uint256 scalingFactor = _getTokenScalingFactor(collateralToken);

    uint256 collateralRecorded;
    uint256 rawCollateralRecorded;
    {
      bytes32 data = poolInfo[pool].collateralData;
      // @audit current number of wstETH tokens held under the contract
(actual collateral)
      collateralRecorded = data.decodeUint(COLLATERAL_BALANCE_OFFSET,
COLLATERAL_DATA_BITS);
      // @audit the number of stETH tokens recorded under the contract
represents the user's actual position assets, including funding costs
that have not yet been charged by the protocol.
      rawCollateralRecorded =
data.decodeUint(RAW_COLLATERAL_BALANCE_OFFSET, RAW_COLLATERAL_DATA_BITS);
    }

    ......

     // recorded data changed, update local cache
      {
        bytes32 data = poolInfo[pool].collateralData;
        // @audit the current number of wstETH tokens held under the
contract (actual collateral), after deducting the number of wstETH tokens
corresponding to the funding cost.
        collateralRecorded = data.decodeUint(COLLATERAL_BALANCE_OFFSET,
COLLATERAL_DATA_BITS);

        // @audit the number of stETH token recorded under the contract
represents the user's actual position assets, after deducting the funding
cost fees of the protocol.
        rawCollateralRecorded =
data.decodeUint(RAW_COLLATERAL_BALANCE_OFFSET, RAW_COLLATERAL_DATA_BITS);
      }
    }
    // @audit calculate the number of stETH tokens corresponding to
wstETH tokens under the protocol based on the current scalingFactor
value.
    rawCollateral = _scaleUp(collateralRecorded, scalingFactor);

    //@audit if there is a difference, it is recorded as protocol
revenue.
    if (rawCollateral > rawCollateralRecorded) {
      unchecked {
        amountRewards = _scaleDown(rawCollateral - rawCollateralRecorded,
scalingFactor);
        _changePoolCollateral(pool, -int256(amountRewards), 0);

        uint256 performanceFeeRewards = (getRewardsExpenseRatio() *
amountRewards) / FEE_PRECISION;
        uint256 harvestBountyRewards = (getHarvesterRatio() *
amountRewards) / FEE_PRECISION;
        pendingRewards += amountRewards - harvestBountyRewards -
performanceFeeRewards;
        performanceFee += performanceFeeRewards;
        harvestBounty += harvestBountyRewards;
      }
    }

  ......
  }
```

With the introduction of the short pool mechanism, users opening short positions borrow wstETH tokens from the long pool contract. Once the debt is repaid, the borrowed wstETH token is returned to the long pool. However, under the current implementation, this flow results in a loss of part of the long pool's revenue—specifically, the portion derived from the increasing exchange rate (`scalingFactor`) between the wstETH token and the stETH token. Furthermore, it may cause a misclassification of long pool users' principal as protocol revenue. The issue is illustrated below.

Assume the current `scalingFactor` is 1.2 (simplified for clarity; the actual on-chain value is scaled by 10^{18}). A user opens a short position and borrows $rawDebtAmount = 12{\times}10^{18}$ stETH token from the short pool, which triggers a withdrawal of wstETH token from the long pool. The actual amount transferred is computed as:

$$scaledAmount = \frac{rawDebtAmount}{scalingFactor} = \frac{12 \times 10^{18}}{1.2} = 10 \times 10^{18} \qquad (8)$$

Thus, the short pool receives 10 wstETH tokens from the long pool. Internally, the long pool mints CreditNote tokens corresponding to 12 stETH tokens, and the short pool records the borrower's debt in stETH tokens as well. This means the user will only need to repay 12 stETH tokens to close the position—regardless of future changes in the exchange rate.

Later, suppose the `scalingFactor` increases to 1.25. When the user repays the 12 stETH token debt, the equivalent amount of wstETH token returned is:

$$scaledAmount = \frac{rawDebtAmount}{scalingFactor} = \frac{12 \times 10^{18}}{1.25} = 9.6 \times 10^{18} \tag{9}$$

Only 9.6 wstETH tokens are returned to the long pool, and the corresponding 12 stETH tokens worth of CreditNote tokens are burned. As a result, the short pool effectively borrowed 10 wstETH tokens but repaid only 9.6 wstETH tokens, extracting 0.4 wstETH tokens of unearned exchange rate gains. These funds, originally attributable to long pool users, are no longer recoverable.

Despite this loss, the long pool continues to compute and deduct exchange-rate-based revenue (the second component of protocol earnings) under the assumption that it has retained the full amount of yield from the wstETH-stETH token conversion. In this scenario, that assumption is invalid, and the protocol may end up deducting this amount from the principal of long pool users instead. This leads to an overstatement of protocol revenue and a corresponding understatement of user balances, which is logically incorrect.

```
// @audit https://github.com/AladdinDAO/fx-protocol-contracts-internal/blob/d60fd7af898b981f8b9042c8e28db72f0cb582cc/contracts/core/short/ShortPoolManager.sol#L786

function _handleBorrow(
    address pool,
    address debtToken,
    uint256 debtAmount,
    uint256 rawDebtAmount, // @audit note that this value refers to the number of stETH tokens
    uint256 poolFeeRatio,
    address referral,
    uint256 referralFeeRatio
  ) internal {


  // @audit it is necessary to borrow wstETH tokens from the long pool (since the actual collateral in the long pool is wstETH tokens).
  ILongPoolManager(counterparty).borrow(IShortPool(pool).counterparty(), pool, rawDebtAmount);
```

```
    ......
  }

  // @audit https://github.com/AladdinDAO/fx-protocol-contracts-
internal/blob/d60fd7af898b981f8b9042c8e28db72f0cb582cc/contracts/core/sho
rt/ShortPoolManager.sol#L809-L831
  function _handleRepay(
    address pool,
    address debtToken,
    uint256 debtAmount,
    uint256 rawDebtAmount, // @audit number of stETH tokens actually
repaid to position
    uint256 poolFeeRatio,
    address referral,
    uint256 referralFeeRatio
  ) internal {
    // handle repay fee for protocol and referral, it is safe to use
unchecked here
    unchecked {
      uint256 poolFee = (debtAmount * poolFeeRatio) / FEE_PRECISION;
      uint256 referralFee = (debtAmount * referralFeeRatio) /
REFERRAL_FEE_PRECISION;
      if (poolFee > 0) {
        IERC20(debtToken).safeTransferFrom(_msgSender(),
closeRevenuePool, poolFee);
      }
      if (referralFee > 0) {
        IERC20(debtToken).safeTransferFrom(_msgSender(), referral,
referralFee);
      }

      // @audit transfer wstETH tokens to short pool
      IERC20(debtToken).safeTransferFrom(_msgSender(), address(this),
debtAmount);

    // @audit transfer wstETH tokens under the contract back to the long
pool, while processing the ledger
    ILongPoolManager(counterparty).repay(IShortPool(pool).counterparty(),
pool, rawDebtAmount, 0);
    }
  }
```

```
// @auidt https://github.com/AladdinDAO/fx-protocol-contracts-
internal/blob/d60fd7af898b981f8b9042c8e28db72f0cb582cc/contracts/core/Poo
lManager.sol#L669-L689
```

```
function borrow(
    address longPool,
    address shortPool,
    uint256 amount
  ) external onlyCounterparty onlyRegisteredPool(longPool)
onlyLongShortPair(longPool, shortPool) {
    address creditNote = IShortPool(shortPool).creditNote();
    {
      uint256 rawCollateral =
ILongPool(longPool).getTotalRawCollaterals();
      uint256 borrowed = IERC20(creditNote).totalSupply();
      if ((rawCollateral * shortBorrowCapacityRatio[longPool]) /
PRECISION < borrowed + amount) {
        revert ErrorBorrowExceedCapacity();
      }
    }

    address collateralToken = ILongPool(longPool).collateralToken();
    uint256 scalingFactor = _getTokenScalingFactor(collateralToken);

    // @audit the number of stETH tokens needs to be converted into the
corresponding number of wstETH tokens
    uint256 scaledAmount = _scaleDown(amount, scalingFactor);

    _transferOut(collateralToken, scaledAmount, counterparty);

    // @audit note that the amount of funds borrowed from the short pool
is calculated based on the amount of stETH token.
    ICreditNote(creditNote).mint(address(this), amount);
  }

// @audit https://github.com/AladdinDAO/fx-protocol-contracts-
internal/blob/d60fd7af898b981f8b9042c8e28db72f0cb582cc/contracts/core/Poo
lManager.sol#L692-L708
function repay(
    address longPool,
    address shortPool,
    uint256 amount, // @audit actual amount of stETH tokens processed
    uint256 shortfall
  ) external onlyCounterparty onlyRegisteredPool(longPool)
onlyLongShortPair(longPool, shortPool) {
    address collateralToken = ILongPool(longPool).collateralToken();
    uint256 scalingFactor = _getTokenScalingFactor(collateralToken);
    uint256 scaledAmount = _scaleDown(amount, scalingFactor);
    // @audit transfer the corresponding amount of wstETH tokens from the
short pool.
    _transferFrom(collateralToken, counterparty, address(this),
scaledAmount);

    // @audit burn debt tokens corresponding to the amount of stETH
tokens
    ICreditNote(IShortPool(shortPool).creditNote()).burn(address(this),
amount);

    // @audit the short pool system bad debt is handled here without
consideration.
    if (shortfall > 0) {
      ILongPool(longPool).reduceCollateral(shortfall);
      uint256 scaledShortfall = _scaleUp(shortfall, scalingFactor);
      _changePoolCollateral(longPool, -int256(scaledShortfall), -
int256(shortfall));
    }
  }
```

Status

The development team acknowledged the issue and addressed it in commit d09d2ad. Specifically, the updated implementation modifies the debt accounting logic in the short pool. Instead of recording user debt in stETH token terms, the system now tracks the exact amount of wstETH token borrowed by short pool users. This change eliminates the exchange rate discrepancy between wstETH and stETH tokens, thereby preventing unintended loss of long pool yield and ensuring that protocol revenue and user principal are correctly accounted for.

4.3.8 The long pool uses an incorrect function, `_scaleUp()`, to calculate the amount of actual collateral to be deducted when covering bad debt from the short pool.

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

Location

PoolManager.sol#L705

Description

When the short pool incurs bad debt and the reserve fund lacks sufficient funds to cover it, the remaining unbacked debt—represented in the code as `shortfall`—must be absorbed by long pool users. Since the short pool tracks user debt in terms of stETH token amounts, the computed `shortfall` also corresponds to an amount denominated in stETH tokens.

However, in the `repay()` function of the long pool contract, the protocol mistakenly applies the `_scaleUp()` function when converting the `shortfall` (in stETH token) to determine how much wstETH token to deduct from the long pool. This is incorrect. In the f(x) protocol, `_scaleUp()` is intended for converting wstETH token amounts into stETH tokens. At the same time, `_scaleDown()` should be used to convert stETH token amounts into their wstETH token equivalents. Therefore, when computing the `scaledShortfall`, the correct approach is to use `_scaleDown()` to ensure the long pool only transfers the appropriate amount of wstETH token corresponding to the stETH-denominated bad debt.

```
// @audit https://github.com/AladdinDAO/fx-protocol-contracts-
internal/blob/d60fd7af898b981f8b9042c8e28db72f0cb582cc/contracts/core/sho
rt/ShortPoolManager.sol#L453-L462
function liquidate(
    address pool,
    address receiver,
    uint256 maxDebt
  ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls, uint256 debts) {
    _checkDebtValues(pool, maxDebt);

    // @audit note that when calculating debt within the protocol, the
stETH quantity is used as the basis. The return value `shortfall` here
represents the amount of bad debt stETH tokens in the short pool ledger.
    (bool underCollateral, uint256 shortfall) =
IShortPool(pool).isUnderCollateral();
    if (underCollateral) {
       ......
      // repay user funds + reserve fund
      ILongPoolManager(counterparty).repay(
        IShortPool(pool).counterparty(),
        pool,
        _scaleUp(debts, scalingFactor) + rawReserveFund,
        shortfall  // @audit the long pool bears the system bad debts
generated by the short pool.
      );
      _transferOut(fxUSD, colls, _msgSender());
      IShortPool(pool).kill();
      return (colls, debts);
    }
    ......
  }
```

```
// @audit https://github.com/AladdinDAO/fx-protocol-contracts-
internal/blob/d60fd7af898b981f8b9042c8e28db72f0cb582cc/contracts/core/Poo
lManager.sol#L692-L708
function repay(
    address longPool,
    address shortPool,
    uint256 amount,
    uint256 shortfall // @audit this value is greater than 0 only when
the short pool processing system handles bad debts
  ) external onlyCounterparty onlyRegisteredPool(longPool)
onlyLongShortPair(longPool, shortPool) {
    address collateralToken = ILongPool(longPool).collateralToken();
    uint256 scalingFactor = _getTokenScalingFactor(collateralToken);
    uint256 scaledAmount = _scaleDown(amount, scalingFactor);
    _transferFrom(collateralToken, counterparty, address(this),
scaledAmount);
    ICreditNote(IShortPool(shortPool).creditNote()).burn(address(this),
amount);

    //@audit `shortfall` refers to the number of stETH tokens.
    if (shortfall > 0) {
      // @audit all long pool users bear the system bad debts incurred by
the short pool.
      ILongPool(longPool).reduceCollateral(shortfall);

      // @audit should use the _scaleDown() function instead of the
_scaleUp() function
      uint256 scaledShortfall = _scaleUp(shortfall, scalingFactor);
      _changePoolCollateral(longPool, -int256(scaledShortfall), -
int256(shortfall));
    }
  }
```

Status

The development team has revised the debt accounting logic in commit d09d2ad. In the updated implementation, the CreditNote token now uses the same number of decimals as the corresponding pool's debt token. This change ensures consistent precision in debt tracking and effectively resolves the issue.

4.3.9 Discussion of the logic of the `_transferCollateralOut()` function.

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

Location

PoolManager.sol#L1113-L1118

Description

Analysis of the long pool contract reveals that the `_transferCollateralOut()` function is responsible for transferring wstETH tokens to the caller when redeeming collateral (or potentially to the referral address as a reward). Since the wstETH token collateral in the long pool may be borrowed by the short pool to support short positions, the long pool might not always have sufficient wstETH token available for redemption. To address this issue, the f(x) protocol introduces the CreditNote token. When the short pool borrows wstETH token collateral from the long pool, the protocol mints an equivalent amount of CreditNote tokens to the long pool. The amount is calculated based on the real-time exchange rate between the wstETH token and the stETH token at the time of borrowing.

If a user attempts to redeem wstETH token collateral and the long pool has insufficient liquidity in wstETH tokens, the user receives CreditNote tokens instead. These tokens can later be redeemed in the short pool for the corresponding amount of assets, which are settled in fxUSD tokens.

```
function redeem(
    address pool,
    uint256 debts,
    uint256 minColls
  ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls) {
    ......

    // @audit the number of stETH tokens redeemed by users. since the
collateral is wstETH tokens, they must be further converted into wstETH
tokens.
    uint256 rawColls = ILongPool(pool).redeem(debts);

    address collateralToken = ILongPool(pool).collateralToken();
    uint256 scalingFactor = _getTokenScalingFactor(collateralToken);

    // @audit the number of stETH tokens is converted into wstETH tokens.
    colls = _scaleDown(rawColls, scalingFactor);
    _changePoolCollateral(pool, -int256(colls), -int256(rawColls));
    _changePoolDebts(pool, -int256(debts));

    // @dev use unchecked to reduce code size and gas
    uint256 protocolFees;
    unchecked {
      protocolFees = (colls * getRedeemFeeRatio()) / FEE_PRECISION;
      _accumulatePoolMiscFee(pool, protocolFees);
      colls -= protocolFees;
    }
    if (colls < minColls) revert ErrorInsufficientRedeemedCollateral();

    // @audit the wstETH tokens will be sent to the caller here.
    //        note that the parameter colls refers to the number of
wstETH tokens.
    _transferCollateralOut(pool, collateralToken, scalingFactor, colls,
_msgSender());
    IFxUSDRegeneracy(fxUSD).burn(_msgSender(), debts);

    emit Redeem(pool, colls, debts, protocolFees);
  }


function _transferCollateralOut(
    address pool,
    address collateralToken,
    uint256 scalingFactor,
    uint256 scaledAmount,  // @audit wstETH token amount
    address receiver
  ) internal {
    address shortPool = ILongPool(pool).counterparty();
    // check if current collateral is enough, if not use debt token to
cover
    if (shortPool != address(0)) {
      address creditNote = IShortPool(shortPool).creditNote();

      // @audit total collateral amount calculated based on the number of
stETH tokens under the current long pool contract.
      uint256 rawCollateral = ILongPool(pool).getTotalRawCollaterals();

      // @audit the number of stETH tokens borrowed from the long pool by
the short pool.
      uint256 borrowed = IERC20(creditNote).balanceOf(address(this));

      // @audit ambiguous code
      if (rawCollateral < borrowed) {
        // not enough collateral, use debt token to cover
        uint256 shortfall = borrowed - rawCollateral;

        uint256 shortfallScaled = _scaleDown(shortfall, scalingFactor);

        // @audit ambiguous code
        IERC20(creditNote).safeTransfer(receiver, shortfall);
        // @audit ambiguous code
        _transferOut(creditNote, scaledAmount - shortfallScaled,
receiver);


      } else {
        _transferOut(collateralToken, scaledAmount, receiver);
      }
    } else {
      _transferOut(collateralToken, scaledAmount, receiver);
    }
  }
```

The `_transferCollateralOut()` function in the long pool contract is responsible for redeeming collateral (wstETH tokens) to users. When handling insufficient funds for wstETH tokens in the long pool, the following logic was used:

```
// @audit total collateral amount calculated based on the number of stETH
tokens under the current long pool contract.
 uint256 rawCollateral = ILongPool(pool).getTotalRawCollaterals();
 // @audit the number of stETH tokens borrowed from the long pool by the
short pool.
 uint256 borrowed = IERC20(creditNote).balanceOf(address(this));

 if (rawCollateral < borrowed) {
    // not enough collateral, use debt token to cover
    uint256 shortfall = borrowed - rawCollateral;
    uint256 shortfallScaled = _scaleDown(shortfall, scalingFactor);
    IERC20(creditNote).safeTransfer(receiver, shortfall);
    _transferOut(creditNote, scaledAmount - shortfallScaled, receiver);
} else {
    // @audit there may not be enough wstETH tokens under the contract.
    _transferOut(collateralToken, scaledAmount, receiver);
}
```

The parameter `rawCollateral` represents the amount of stETH tokens provided as user collateral in the long pool after deducting the accrued funding cost. The parameter `borrowed` denotes the amount of stETH tokens borrowed by the short pool from the long pool. We have identified the following three concerns:

(1). The condition `if (rawCollateral < borrowed)` appears to be insufficient for ensuring correct collateral availability. This condition can be triggered in scenarios where the short pool has borrowed a large amount of wstETH from the long pool. Over time, as funding costs are collected from the long pool, the `rawCollateral` value gradually decreases and may fall below the `borrowed` amount. In such cases, it effectively means that a portion of the long pool's funding cost is being lent to the short pool. As a result, the long pool would no longer have sufficient wstETH tokens to fulfill user redemption requests, and users should receive an equivalent amount of CreditNote tokens instead. Moreover, even when `rawCollateral >= borrowed`, a redemption failure could still occur if the expected redemption amount (`scaledAmount`) exceeds the available token balance in the long pool (`rawCollateral - borrowed`). This case is not explicitly checked in the `else` branch, which proceeds to call `_transferOut()` to send wstETH tokens to the user. If the contract holds an insufficient amount of wstETH tokens, this transfer may fail.

(2). Based on the above, the difference `borrowed - rawCollateral`, represented as `shortfall`, can be interpreted as the portion of the funding cost from the long pool that has effectively been lent to the short pool. This `shortfall` amount should correspond to the long pool's funding cost contribution and, accordingly, the long pool should be credited with an equivalent amount of CreditNote tokens. However, the current implementation directly transfers the `shortfall` amount of CreditNote tokens to the caller, which raises questions and requires clarification.

(3). The expression `scaledAmount - shortfallScaled` is used in the code and appears to represent an amount of wstETH tokens. However, given that CreditNote tokens in the f(x) protocol are designed to correspond directly to stETH token amounts, this approach may be inconsistent and could require further review.

Status

The developer team refactored this part of the code in commit bd6e772 and fixed the above issues.

4.3.10 It is recommended to remove the conditional check for the `_checkValueTooLarge()` function within the `_updateMiscData()` function to save gas.

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

Location

ProtocolFees.sol#L331

Description

The `_checkValueTooLarge()` condition within the `_updateMiscData()` function appears to be redundant, as all invocations of `_updateMiscData()` already validate the input parameters beforehand. Removing this internal check could lead to minor gas savings.

```
/// @dev The maximum expense ratio.
uint256 private constant MAX_EXPENSE_RATIO = 5e8; // 50%

/// @dev The maximum harvester ratio.
uint256 private constant MAX_HARVESTER_RATIO = 2e8; // 20%


/// @dev The maximum flash loan fee ratio.
uint256 private constant MAX_FLASH_LOAN_FEE_RATIO = 1e8; // 10%

/// @dev The maximum redeem fee ratio.
uint256 private constant MAX_REDEEM_FEE_RATIO = 1e8; // 10%


function _updateMiscData(uint256 newValue, uint256 offset, uint256
length) private returns (uint256 oldValue) {
    // @audit this condition seems redundant, as all external calls to
this function have been checked.
    _checkValueTooLarge(newValue, MAX_EXPENSE_RATIO);

    bytes32 _data = _miscData;
    oldValue = _miscData.decodeUint(offset, length);
    _miscData = _data.insertUint(newValue, offset, length);

    return oldValue;
}


function _updateRewardsExpenseRatio(uint256 newRatio) private {
    // @audit newRatio has been evaluated.
    _checkValueTooLarge(newRatio, MAX_EXPENSE_RATIO);
    uint256 oldRatio = _updateMiscData(newRatio,
REWARDS_EXPENSE_RATIO_OFFSET, 30);
    emit UpdateRewardsExpenseRatio(oldRatio, newRatio);
}

/// @dev Internal function to update the fee ratio distributed to
treasury.
/// @param newRatio The new ratio to update, multiplied by 1e9.
function _updateLiquidationExpenseRatio(uint256 newRatio) private {
  // @audit newRatio has been evaluated.
  _checkValueTooLarge(newRatio, MAX_EXPENSE_RATIO);
  uint256 oldRatio = _updateMiscData(newRatio,
LIQUIDATION_EXPENSE_RATIO_OFFSET, 30);
  emit UpdateLiquidationExpenseRatio(oldRatio, newRatio);
}

/// @dev Internal function to update the fee ratio distributed to
treasury.
/// @param newRatio The new ratio to update, multiplied by 1e9.
function _updateFundingExpenseRatio(uint256 newRatio) private {
  // @audit newRatio has been evaluated.
  _checkValueTooLarge(newRatio, MAX_EXPENSE_RATIO);
  uint256 oldRatio = _updateMiscData(newRatio,
FUNDING_EXPENSE_RATIO_OFFSET, 30);
  emit UpdateFundingExpenseRatio(oldRatio, newRatio);
}

/// @dev Internal function to update the fee ratio distributed to
harvester.
/// @param newRatio The new ratio to update, multiplied by 1e9.
function _updateHarvesterRatio(uint256 newRatio) private {
  // @audit newRatio has been evaluated.
  _checkValueTooLarge(newRatio, MAX_HARVESTER_RATIO);
  uint256 oldRatio = _updateMiscData(newRatio, HARVESTER_RATIO_OFFSET,
30);
  emit UpdateHarvesterRatio(oldRatio, newRatio);
}

/// @dev Internal function to update the flash loan fee ratio.
/// @param newRatio The new ratio to update, multiplied by 1e9.
function _updateFlashLoanFeeRatio(uint256 newRatio) private {
    // @audit newRatio has been evaluated.
    _checkValueTooLarge(newRatio, MAX_FLASH_LOAN_FEE_RATIO);
    uint256 oldRatio = _updateMiscData(newRatio, FLASH_LOAN_RATIO_OFFSET,
30);
```

```
    emit UpdateFlashLoanFeeRatio(oldRatio, newRatio);
  }

  /// @dev Internal function to update the redeem fee ratio.
  /// @param newRatio The new ratio to update, multiplied by 1e9.
  function _updateRedeemFeeRatio(uint256 newRatio) private {
    // @audit newRatio has been evaluated.
    _checkValueTooLarge(newRatio, MAX_REDEEM_FEE_RATIO);
    uint256 oldRatio = _updateMiscData(newRatio, REDEEM_FEE_RATIO_OFFSET,
  30);
    emit UpdateRedeemFeeRatio(oldRatio, newRatio);
  }
```

Status

The development team confirmed this issue and removed the redundant judgment in commit d09d2ad.

4.3.11 Logic discussion for the function `onCloseOrRemoveShortPositionFlashLoan()`.

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

Location

ShortPositionOperateFlashLoanFacet.sol#L182

Description

When users do not hold sufficient debt tokens (wstETH token) to close their short positions, they can utilize the protocol's flash loan–based closure mechanism by calling the `closeOrRemoveShortPositionFlashLoan()` function. This function facilitates position closure (or partial collateral redemption) through the following steps:

Step 1: Based on user input, the function initiates a flash loan from the Morpho protocol to borrow the required debt tokens (wstETH).

Step 2: The borrowed wstETH token is then used to close the position (or redeem part of the collateral). At this stage, the collateral, in the form of FXUSD tokens, is transferred to the facet contract. Typically, the value of the redeemed fxUSD tokens exceeds the value of the repaid wstETH token debt.

Step 3: The user specifies a `swapAmount` of fxUSD tokens to be swapped for wstETH tokens, which will be used to repay the flash loan.

Step 4: The remaining fxUSD tokens held in the facet contract are returned to the user. Any remaining wstETH tokens in the facet contract are sent to the `revenuePool` address.

Given this flow, it is important to note that the `swapAmount` parameter is explicitly provided by the user. Since the value of the fxUSD token obtained from closing the position usually exceeds the value of the wstETH token debt, if a user chooses to swap all of their received fxUSD tokens for wstETH tokens, any remaining wstETH tokens (originating from the user's original collateral) will be sent to the `revenuePool` address. The user will receive no residual funds. This behavior should be confirmed.

```
function closeOrRemoveShortPositionFlashLoan(
    LibRouter.ConvertOutParams memory params,
    address pool,
    uint256 positionId,
    uint256 fxUSDWithdrawAmount,
    uint256 debtTokenBorrowAmount,
    bytes calldata data
  ) external nonReentrant {
    // flashloan debt token
    // repay debt token and get fxUSD back
    // swap fxUSD to debt token to repay flashloan
    address debtToken = IShortPool(pool).debtToken();
    _invokeFlashLoan(
      debtToken,
      debtTokenBorrowAmount,
      abi.encodeCall(

 ShortPositionOperateFlashLoanFacet.onCloseOrRemoveShortPositionFlashLoan
,
        (pool, positionId, fxUSDWithdrawAmount, debtTokenBorrowAmount,
msg.sender, data)
      )
    );

    // convert all fxUSD to target token
    LibRouter.convertAndTransferOut(params, fxUSD,
IERC20(fxUSD).balanceOf(address(this)), msg.sender);

    // @audit all remaining wstETH token under the contract will be sent
to the revenuePool
    // transfer extra debt token to revenue pool
    LibRouter.refundERC20(debtToken,
LibRouter.routerStorage().revenuePool);
}


function onCloseOrRemoveShortPositionFlashLoan(
    address pool,
    uint256 position,
    uint256 fxUSDWithdrawAmount,
    uint256 debtTokenBorrowAmount,
    address recipient,
    bytes memory data
  ) external onlySelf {

    // @audit the swapAmount parameter is provided by the caller and
specifies the amount of fxUSD token to be redeemed for wstETH tokens
using the swapAmount, in order to repay the flash loan.
    (bytes32 miscData, uint256 swapAmount, address swapTarget, bytes
memory swapData) = abi.decode(
      data,
      (bytes32, uint256, address, bytes)
    );

    address debtToken = IShortPool(pool).debtToken();
    LibRouter.approve(debtToken, poolManager, debtTokenBorrowAmount);

  ......

    // swap fxUSD to debt token to repay flashloan
    // @audit exchange the swapAmount quantity of fxUSD tokens into
wstETH tokens to repay the flash loan.
    _swap(fxUSD, debtToken, swapAmount, debtTokenBorrowAmount,
swapTarget, swapData);

    emit PositionOperate(pool, position, int256(fxUSDWithdrawAmount -
swapAmount), 0);
  }
```

Status

The developer team has confirmed this issue. The `revenuePool` address is set by the contract administrator, and the overall risk is controllable.

4.3.12 Adjust the decimals of the `CreditNote` token to suit the actual scenario.

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

Location

CreditNote.sol#L32-L34

Description

According to the latest code implementation, CreditNote tokens are expected to have the same number of decimals as the corresponding pool's debt token. For example, if the debt token is WBTC, the associated CreditNote token should use 8 decimals. However, the default `ERC20Upgradeable` contract from OpenZeppelin sets the decimals to 18 by default (see: OpenZeppelin ERC20Upgradeable.sol#L98). This discrepancy should be addressed to ensure that the CreditNote token decimals accurately reflect those of the underlying debt token.

```
import { ERC20Upgradeable } from "@openzeppelin/contracts-
upgradeable/token/ERC20/ERC20Upgradeable.sol";

import { ICreditNote } from "../../interfaces/ICreditNote.sol";

contract CreditNote is ERC20Upgradeable, ICreditNote {

  ......

constructor(address _poolManager) {
    poolManager = _poolManager;
  }

  function initialize(string memory name, string memory symbol) external
initializer {
    __ERC20_init(name, symbol);
  }

 ......
}
```

Status

The development team has acknowledged this issue and addressed it in commit b681c16 by updating the `initialize()` function. The function now includes a parameter to set the `decimals` value during initialization. This allows the protocol administrator to configure the correct number of decimals for each CreditNote token at the time of protocol deployment.

4.3.13 Discussion on the fee mechanism for redeeming fxUSD tokens via the `redeemByCreditNote()` function in the `ShortPoolManager` contract.

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

Location

ShortPoolManager.sol#L316

Description

Users can call the `redeemByCreditNote()` function to convert their CreditNote tokens into fxUSD tokens. According to the implementation (link), a fee is charged based on the amount of fxUSD token being redeemed. CreditNote tokens are issued when the short pool borrows wstETH tokens from the long pool, causing the long pool's actual wstETH tokens holdings to fall short of the total user collateral. When a long pool user attempts to redeem, rebalance, or liquidate their position and the protocol lacks sufficient wstETH tokens, they receive CreditNote tokens instead. These tokens can later be redeemed through the short pool for fxUSD tokens.

However, it is important to note that the long pool already charges a fee when users perform redemption-related actions such as `redeem`, `rebalance`, or `liquidate`. Therefore, applying an additional fee during the `redeemByCreditNote()` process results in double-charging the user. For the current implementation, this can be resolved by setting the short pool's redeem fee ratio to 0 at protocol launch, while retaining the fee interface for future flexibility.

```
function redeemByCreditNote(
    address pool,
    uint256 debts,
    uint256 minColls
  ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls) {
    ......

    _changePoolCollateral(pool, -int256(colls));
    _changePoolRawDebts(pool, -int256(rawDebts));

    uint256 protocolFees = (colls * getRedeemFeeRatio()) / FEE_PRECISION;
    _accumulatePoolMiscFee(pool, protocolFees);
    colls -= protocolFees;
    if (colls < minColls) revert ErrorInsufficientRedeemedCollateral();
    _transferOut(fxUSD, colls, _msgSender());

 ILongPoolManager(counterparty).repayByCreditNote(IShortPool(pool).counte
rparty(), pool, debts);

    emit RedeemByCreditNote(pool, colls, debts, protocolFees);
  }
```

Status

The development team has acknowledged the issue and stated that the fee rate will be set to 0 when the protocol is deployed to production, thereby preventing users from being charged fees redundantly.

4.3.14 High-leverage user positions may be reduced as a result of collateral redemption actions.

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

Location

ShortPoolManager.sol#L295-L325

PoolManager.sol#L390-L427

Description

Similar to how redeeming fxUSD tokens from the long pool can reduce the leverage of highly leveraged users within that pool, redeeming fxUSD tokens from the short pool using CreditNote tokens can also reduce the leverage of highly leveraged users in the short pool. Moreover, suppose a user borrows a large amount of fxUSD tokens and redeems them in the long pool for wstETH tokens and some CreditNote tokens (assuming the long pool holds insufficient wstETH tokens), and then uses the received CreditNote tokens to redeem fxUSD tokens from the short pool. In that case, this process can simultaneously reduce leverage in both pools. Such behavior may have amplified effects on the protocol's overall leverage dynamics. To mitigate this, an appropriate fee rate should be configured for long pool redemptions at the time of protocol deployment.

```
function redeemByCreditNote(
    address pool,
    uint256 debts,
    uint256 minColls
```

```
  ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls) {
    address creditNote = IShortPool(pool).creditNote();
    IERC20(creditNote).safeTransferFrom(_msgSender(), address(this),
debts);

    address debtToken = IShortPool(pool).debtToken();
    uint256 scalingFactor = _getTokenScalingFactor(debtToken);
    uint256 rawDebts = _scaleUp(debts, scalingFactor);
    if (rawDebts > IShortPool(pool).getTotalRawDebts()) {
      revert ErrorRedeemExceedTotalDebt();
    }
    _checkRawDebtValues(pool, rawDebts);

    colls = IShortPool(pool).redeemByCreditNote(rawDebts);

    ......

}

// @audit https://github.com/AladdinDAO/fx-protocol-contracts-
internal/blob/f107794dd5ea57ab1aaa7dc0b34e989fab918051/contracts/core/Poo
lManager.sol#L390
function redeem(
    address pool,
    uint256 debts,
    uint256 minColls
  ) external onlyRegisteredPool(pool) nonReentrant whenNotPaused returns
(uint256 colls) {
    if (debts > IERC20(fxUSD).balanceOf(_msgSender())) {
      revert ErrorRedeemExceedBalance();
    }
    if (debts < MIN_REDEEM_DEBTS) {
      revert ErrorRedeemDebtsTooSmall();
    }
    if (!IPoolConfiguration(configuration).isRedeemAllowed()) {
      revert ErrorRedeemNotAllowed();
    }

    uint256 rawColls = ILongPool(pool).redeem(debts);

    address collateralToken = ILongPool(pool).collateralToken();
    uint256 scalingFactor = _getTokenScalingFactor(collateralToken);
    colls = _scaleDown(rawColls, scalingFactor);
```

```
        _changePoolCollateral(pool, -int256(colls), -int256(rawColls));
        _changePoolDebts(pool, -int256(debts));

      ......
    }
```

Status

According to the development team, after the short pool is launched, the long pool's maximum debt exposure is capped at 50%. Therefore, in scenarios where the long pool has exhausted its wstETH tokens, it typically indicates a market downturn in which many long positions are being closed. Under such conditions, fxUSD tokens are expected to trade at a premium. Consequently, redeeming fxETH tokens into fxUSD tokens and selling them on the secondary market is a rational behavior, and it is unlikely that users would re-enter the long pool to redeem fxUSD tokens again. Additionally, the f(x) protocol explicitly disables the `redeem()` function when the fxUSD token is trading at a premium. Taken together, this means that typical market behavior, such as the closure of a long position, will generally be accompanied by a one-time reduction in the leverage of a highly leveraged short position. This process is not expected to result in recursive leverage reduction across positions. Therefore, the associated risk is considered to be within a controllable range.

5. Conclusion

After auditing and analyzing the f(x) Protocol 2.1, 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.