Security Audit Report

f(x) Protocol 2.0 by AladdinDAO

SECBIT

January 1, 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. Building upon the foundation of its predecessor v1.0, f(x) Protocol's version 2.0 enhances its core functionalities. The protocol retains its stablecoin component (fxUSD) while improving the leveraged trading feature (xPOSITION). This upgraded version eliminates individual liquidation risks and funding fees while providing high fixed-leverage trading opportunities. Furthermore, version 2.0 emphasizes user-friendly operations through improved stability mechanisms and a more refined interface. SECBIT Labs conducted an audit from November 13, 2024 to January 1, 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 2.0 contract update 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 Significant precision deviation exists in calculating the `aaveRatio` parameter.	Info	Fixed
Design & Implementation	4.3.2 Asynchronous rate parameter updates between the FX protocol and Aave protocol lending rates directly affect funding cost computation mechanisms.	Info	Fixed
Design & Implementation	4.3.3 The current protocol requires users to actively invoke the `liquidate()` function to process position bad debt, potentially resulting in improper debt allocation.	Low	Discussed
Design & Implementation	4.3.4 Recommend implementing a validation check within the `_liquidateTick()` function to verify if the calculated `newTick` exceeds the `topTick` value, preventing incorrect maximum tick updates.	Info	Discussed
Design & Implementation	4.3.5 Vulnerability in the `rebalance()` function due to the absence of maximum debt ratio limitations.	Low	Fixed
Design & Implementation	4.3.6 When liquidating a position, special attention must be given to setting the maximum liquidation debt amount `maxRawDebts` to prevent potential liquidation failures.	Info	Fixed
Design & Implementation	4.3.7 Discussion of the `rebalance()` function logic.	Info	Discussed
Design & Implementation	4.3.8 Frequent swapping between `fxusd` tokens and `usdc` tokens within the protocol may lead to the erosion of the total asset value held by the protocol.	Low	Discussed
Design & Implementation	4.3.9 When using the `onMigrateXstETHPosition()` function to migrate positions, there is a possibility that the minted `fxusd` token debt will be insufficient to repay the borrowed `usdc` token amount, which may result in the migration failing.	Info	Discussed
Design & Implementation	4.3.10 The remaining funds in the contract should be sent to the caller.	Info	Discussed
Design & Implementation	4.3.11 Bypassing the `flashLoanContext` check in the `FlashLoanCallbackFacet` contract could allow an attacker to exploit the contract logic.	Medium	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the f(x) Protocol 2.0 update is shown below:

- Smart contract code
    - initial review commit *fcf3fa6*
    - final review commit *c3c45b5*

2.2 Contract List

The following content shows the contracts included in the f(x) Protocol 2.0, which the SECBIT team audits:

Name	Lines	Description
AaveFundingPool.sol	162	The supplementary module of the protocol updates the collateral index system, which is utilized for calculating and collecting funding costs.
BasePool.sol	382	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.
PoolConstant.sol	15	The auxiliary component of the protocol documents the predetermined constant values implemented throughout the protocol's operational architecture.
PoolErrors.sol	29	The auxiliary component of the protocol documents the enumerated error states and their associated failure conditions encountered during protocol operations.
PoolStorage.sol	205	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	156	The supplementary module of the protocol calculates corresponding tick

		values based on position debt ratios, facilitating position liquidation processes through tick-based execution mechanisms.
FlashLoans.sol	42	The supplementary module of the protocol implements flash loan functionality, enabling users to access protocol-controlled capital through instantaneous borrowing mechanisms.
FxUSDBasePool.sol	410	Users can deposit fxUSD and USDC tokens into the protocol contract, where these assets are utilized for position leverage adjustment and protocol risk mitigation.
FxUSDRegeneracy.sol	313	The protocol processes legacy issues from previous f(x) protocol versions, enabling the conversion of fTokens to base tokens through designated rebalance pools.
PegKeeper.sol	120	The supplementary module of the protocol maintains stability in the exchange ratio between fxUSD tokens and USDC tokens.
PoolManager.sol	402	The peripheral contract provides direct interfaces for users to execute position operations, including position opening, closing, adjustment, and liquidation procedures.
ProtocolFees.sol	160	The contract configures protocol fee parameters, including position opening fees, closing fees, liquidation fees, and other associated cost metrics.
ReservePool.sol	53	The contract temporarily stores protocol subsidy funds, which supplement liquidation rewards when the standard liquidation incentives are insufficient for liquidating users.
GaugeRewarder.sol	32	Harvest base token to target token by amm trading and distribute to fxBASE gauge.
TickBitmap.sol	34	The library contract implements tick initialization and query functionalities, incorporating design principles adapted from Uniswap V3's architectural logic.
TickMath.sol	172	The library contract computes tick values and ratioX96 parameters based on provided debt obligations and collateral quantities.
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.
MigrateFacet.sol	175	The peripheral contract facilitates user migration of legacy xToken holdings to fixed-leverage xPosition instruments in the new protocol version.
PositionOperateFlashLoanFacet.sol	130	The peripheral contract provides interfaces enabling users to convert their deposited collateral into xPosition positions fully.

RouterManagementFacet.sol	48	The parameter configuration contract governing facet-specific operational settings within the protocol's diamond architecture.
LSDPriceOracleBase.sol	100	The Oracle abstract contract aggregates asset price data from multiple protocols, processing and providing consolidated price feeds for protocol utilization.
SpotPriceOracleBase.sol	84	The Oracle base contract aggregates spot price data from multiple protocols for asset valuation.
StETHPriceOracle.sol	21	The contract provides stETH asset price data feeds for protocol implementation.

3. Contract Analysis

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

3.1 Role Classification

Three key roles in the f(x) Protocol 2.0 are Governance Account and Common Account.

- Governance Account
  - Description

    Contract Administrator
  - Authority
    - Update fees ratio
    - Update market and incentive configurations
    - Transfer ownership
    - Pause crucial functions
  - 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 2.0
  - Authority

        - Open and close position using authorized tokens, resulting in fixed leverage allocation
        - Redeem base tokens with fxusd tokens
        - 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

The f(x) protocol implements a decentralized quasi-stablecoin with high collateral utilization efficiency and leveraged contracts with low liquidation risks and no funding costs. The new upgrade preserves the stablecoin system (fxUSD) from the previous version while substantially improving the leveraged trading capability (xPOSITION). Users can now engage in high fixed-leverage trading activities without concerns about individual liquidations or funding fee charges. 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 five parts:

PoolManager

The contract provides core protocol interfaces for users, encompassing position opening, closing, adjustment, liquidation operations, and protocol funding cost collection mechanisms. The main functions in this contract are as follows:

- `operate()`

  The function enables users to initiate positions and execute position adjustments.
- `redeem()`

  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 any user to distribute rewards through this function while receiving a proportional share of distribution incentives.

FxUSDBasePool

The contract implements market stabilization mechanisms through fxUSD token absorption, maintaining price equilibrium between fxUSD and USDC token pairs. The main functions in this contract are as follows:

- `deposit()`

  Through this function, any user can deposit fxUSD tokens or USDC tokens and receive proportional fxUSD base pool token shares in return.

- `redeem()`

  Users can redeem their shares through this function based on the current ratio of fxUSD tokens and USDC tokens within the contract.

- `rebalance()`

  When protocol position debt ratios reach the rebalance threshold, users can invoke this function to redeem position debt and reduce debt ratios, primarily utilizing fxUSD tokens held within the contract.

- `liquidate()`

  When protocol position debt ratios reach the liquidation threshold, users can invoke this function to execute position liquidation, primarily utilizing fxUSD tokens held within the contract.

- `arbitrage()`

  Administrators can utilize this function to execute secondary market trades between fxUSD tokens and USDC tokens, maintaining fxUSD exchange rate stability.

FxUSDBasePoolFacet

The auxiliary contract for `FxUSDBasePool` facilitates user conversion of external assets into fxUSD tokens or USDC tokens for subsequent deposit into the `FxUSDBasePool` contract. The main functions in this contract are as follows:

- `migrateToFxBase()`

  Users holding legacy assets can utilize this function to withdraw and convert their holdings into fxUSD tokens, which they can then deposit into the `FxUSDBasePool` contract.

- `migrateToFxBaseGauge()`

  Users holding legacy assets can utilize this function to withdraw and convert their holdings into fxUSD tokens for deposit into the `FxUSDBasePool` contract. Subsequently, the received fxUSD base pool token shares are deposited into corresponding gauges to earn FXN token rewards.

- `depositToFxBase()`

  Users can convert their external assets into fxUSD tokens for subsequent deposit into the fxUSD base pool.

- `depositToFxBaseGauge()`

  Users can convert their external assets into fxUSD tokens, which they deposit into the fxUSD base pool. Subsequently, the received shares are deposited into corresponding gauges to earn FXN token rewards.

- `redeemFromFxBase()`

  Users can burn their fx base pool shares to retrieve corresponding fxUSD tokens and USDC tokens.

- `redeemFromFxBaseGauge()`

  Users can withdraw fx base pool shares from gauges and burn them to retrieve fxUSD tokens and USDC tokens from the `FxUSDBasePool` contract.

MigrateFacet

The protocol provides this contract to rapidly convert user-held xTokens into fixed-leverage xPOSITION instruments in the new protocol version. The main functions in this contract are as follows:

- `migrateXstETHPosition()`

  The function leverages flash loan mechanisms to enable users to convert their xstETH tokens into xPOSITION fully.

- `migrateXfrxETHPosition()`

  The function leverages flash loan mechanisms to enable users to fully convert their xfrxETH tokens into xPOSITION.

PositionOperateFlashLoanFacet

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

- `openOrAddPositionFlashLoan()`

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

- `closeOrRemovePositionFlashLoan()`

  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 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 Significant precision deviation exists in calculating the `aaveRatio` parameter.

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

Location

AaveFundingPool.sol#L139

Description

The protocol's position opening fees utilize tiered rates for calculation, with the tier rate coefficient `aaveRatio` referencing lending rates from the Aave protocol. Using data from the USDC token on Ethereum within Aave protocol as an example (see: aave.getReserveData()), the real-time value of parameter `reserveData.currentVariableBorrowRate` is 195786379192399030579331133, resulting in a calculated `rate` of approximately 1.957×10^{17}.

The protocol's `openRatioStep` parameter is set to 5×10^{16}. Given the minimal difference between these parameters, division operations result in significant precision loss. From a protocol fee perspective, consider implementing ceiling rounding or applying precision coefficients to minimize calculation errors.

Additionally, given the substantial value of the `rate` parameter, the subtraction of 1 appears unnecessary.

```
function initialize(
    address admin,
    string memory name_,
    string memory symbol_,
    address _collateralToken,
    address _priceOracle
  ) external initializer {
    ......

    _grantRole(DEFAULT_ADMIN_ROLE, admin);

    // @audit During initialization, openRatio is set to 1e6 and
openRatioStep is configured to 5e16.
    _updateOpenRatio(1000000, 50000000000000000); // 0.1% and 5%

    _updateCloseFeeRatio(1000000); // 0.1%
    _updateInterestRate();
  }
```

```
function getOpenFeeRatio() public view returns (uint256) {
     // @audit During initialization, openRatioStep is configured to 5e16.
    (uint256 openRatio, uint256 openRatioStep) = _getOpenRatio();

    // @audit According to comments and on-chain data, the converted rate value ranges approximately between the magnitude of 1e16 and 1e18.
    (uint256 rate, ) = _getInterestRate();
    unchecked {
    uint256 aaveRatio = rate <= openRatioStep ? 1 : (rate - 1) / openRatioStep;
    return aaveRatio * openRatio;
  }
}


/// @dev Internal function to get open ratio and open ratio step.
/// @return ratio The value of open ratio, multiplied by 1e9.
/// @return step The value of open ratio step, multiplied by 1e18.
function _getOpenRatio() internal view returns (uint256 ratio, uint256 step) {
  bytes32 data = fundingMiscData;
  ratio = data.decodeUint(OPEN_RATIO_OFFSET, 30);
  step = data.decodeUint(OPEN_RATIO_STEP_OFFSET, 60);
}

/// @dev Internal function to update interest rate snapshot.
function _updateInterestRate() internal {
  IAaveV3Pool.ReserveDataLegacy memory reserveData = IAaveV3Pool(lendingPool).getReserveData(baseAsset);
  // the interest rate from aave is scaled by 1e27, we want 1e18 scale.

  // @audit The interest rate value is updated here, with on-chain data analysis indicating the rate parameter approximates a magnitude range of 1e16 to 1e18.
  uint256 rate = reserveData.currentVariableBorrowRate / 1e9;
  if (rate > MAX_INTEREST_RATE) rate = MAX_INTEREST_RATE;

  bytes32 data = fundingMiscData;
  data = data.insertUint(rate, INTEREST_RATE_OFFSET, 68);
  fundingMiscData = data.insertUint(block.timestamp, TIMESTAMP_OFFSET, 36);

  emit SnapshotAaveInterestRate(rate, block.timestamp);
  }
```

Status

The development team modified the relevant code logic, replacing the instantaneous rate with the average borrowing rate to prevent sharp rate fluctuations and fix the issue in commit *324b626*.

4.3.2 Asynchronous rate parameter updates between the FX protocol and Aave protocol lending rates directly affect funding cost computation mechanisms.

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

Location

AaveFundingPool.sol#L257-L277

Description

The lending rates within the Aave protocol undergo real-time adjustments corresponding to user borrowing volumes, exhibiting substantial rate escalation during high-volume borrowing events. The protocol implements funding cost updates and captures Aave's `reserveData.currentVariableBorrowRate` parameter during invocations of operational functions (`operate()/redeem()/rebalance()/liquidate()`). This captured rate parameter is the basis for funding cost computations during subsequent protocol interactions.

Consider a scenario at time t_0 when large-scale borrowing in the Aave protocol causes lending rates to surge to $rate_0$. If a user executes protocol functions (such as position opening/closing), this elevated $rate_0$ is recorded. Subsequently, if no position operations occur until time t_1, the protocol calculates funding costs for the period (t_1 - t_0) using the high $rate_0$.

The fx protocol's lending rate cannot dynamically adjust to Aave protocol rate fluctuations; instead, it requires user-triggered updates through `operate()/redeem()` function calls, introducing latency. Significant Aave rate volatility may result in substantial funding cost calculation discrepancies:

(1). High Rate Scenario:

- Aave experiences significant borrowing, causing rate spike

- fx protocol user executes position operations
- Users bear elevated funding costs despite potential Aave rate normalization
- Higher rates persist until the next user-triggered update

(2). Low Rate Scenario:

- Aave rates decrease significantly
- fx protocol operations lock in lower rates
- Protocol collects reduced funding costs until the next update

More frequent position operations by FX users result in more accurate funding cost assessments.

```
function _updateCollAndDebtIndex() internal virtual override returns
(uint256 newCollIndex, uint256 newDebtIndex) {
    // @audit Retrieve the most recently updated funding cost
coefficient newCollIndex.
    (newDebtIndex, newCollIndex) = _getDebtAndCollateralIndex();

    // @audit Retrieve the most recently updated interest rate value.
    (uint256 oldInterestRate, uint256 snapshotTimestamp) =
_getInterestRate();
    if (block.timestamp > snapshotTimestamp) {
      if (IPegKeeper(pegKeeper).isFundingEnabled()) {
        (, uint256 totalColls) = _getDebtAndCollateralShares();
        uint256 totalRawColls = _convertToRawColl(totalColls,
newCollIndex, Math.Rounding.Down);

        // @audit Calculate protocol funding amounts using the most
recently recorded interest rate.
        uint256 funding = (totalRawColls * oldInterestRate *
(block.timestamp - snapshotTimestamp)) /
          (365 * 86400 * PRECISION);
        funding = ((funding * _getFundingRatio()) / FEE_PRECISION);

        // update collateral index with funding costs
        newCollIndex = (newCollIndex * totalRawColls) / (totalRawColls -
funding);

        // @audit Update funding cost index
        _updateCollateralIndex(newCollIndex);
      }
```

```
   // @audit Update interest rate
   // update interest snapshot
   _updateInterestRate();
  }
 }

/// @dev Internal function to update interest rate snapshot.
 function _updateInterestRate() internal {
 // @audit This data represents real-time lending metrics with
significant volatility.
  IAaveV3Pool.ReserveDataLegacy memory reserveData =
IAaveV3Pool(lendingPool).getReserveData(baseAsset);
  // the interest rate from aave is scaled by 1e27, we want 1e18
scale.
  uint256 rate = reserveData.currentVariableBorrowRate / 1e9;
  if (rate > MAX_INTEREST_RATE) rate = MAX_INTEREST_RATE;

  bytes32 data = fundingMiscData;
  data = data.insertUint(rate, INTEREST_RATE_OFFSET, 68);
  fundingMiscData = data.insertUint(block.timestamp, TIMESTAMP_OFFSET,
36);

  emit SnapshotAaveInterestRate(rate, block.timestamp);
 }
```

Status

The development team modified the relevant code logic, replacing the instantaneous rate with the average borrowing rate to prevent sharp rate fluctuations and fix the issue in commit *324b626*.

4.3.3 The current protocol requires users to actively invoke the `liquidate()` function to process position bad debt, potentially resulting in improper debt allocation.

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

Location

AaveFundingPool.sol#L257-L277

BasePool.sol#L94

BasePool.sol#L196

BasePool.sol#L251

BasePool.sol#L293

Description

The FX protocol design implements two indices: the coll index and the debt index. The coll index enables the calculation of net collateral value after funding cost deduction, while the debt index calculates actual debt (fxUSD) obligations after the system's bad debt distribution.

The `_updateCollAndDebtIndex()` function only updates the coll index parameter, leaving the debt index unchanged. Notably, the debt index parameter updates occur exclusively when users invoke the `liquidate()` function to process position liquidations resulting in bad debt.

When protocol positions incur bad debt without immediate processing (pending user invocation of `liquidate()` function), system-wide bad debt remains undistributed among position holders. This creates two scenarios:

(1). New Position/Debt Scenario: Users invoking the `operate()` function to open positions or increase debt (minting fxUSD tokens) may subsequently bear disproportionate system debt when bad debt processing occurs. These users assume responsibility for pre-existing bad debt that existed prior to their position creation, resulting in inequitable debt distribution.

(2). Debt Reduction/Position Closure Scenario: Users invoking the `redeem()` function for debt repayment (burning fxUSD tokens for LSD redemption), the `rebalance()` function for position adjustment, or position closure may evade bad debt responsibility if system bad debt processing occurs subsequently. This design vulnerability enables strategic debt evasion, increasing loss allocation to remaining users.

```
// @audit Only the coll index is updated here, while the debt index
remains unchanged.
function _updateCollAndDebtIndex() internal virtual override returns
(uint256 newCollIndex, uint256 newDebtIndex) {
     // @audit The newDebtIndex parameter is retrieved but remains
unmodified in the subsequent operations.
    (newDebtIndex, newCollIndex) = _getDebtAndCollateralIndex();

    (uint256 oldInterestRate, uint256 snapshotTimestamp) =
_getInterestRate();
    if (block.timestamp > snapshotTimestamp) {
      if (IPegKeeper(pegKeeper).isFundingEnabled()) {
        (, uint256 totalColls) = _getDebtAndCollateralShares();
        uint256 totalRawColls = _convertToRawColl(totalColls, newCollIndex, Math.Rounding.Down);
        uint256 funding = (totalRawColls * oldInterestRate * (block.timestamp - snapshotTimestamp)) /
          (365 * 86400 * PRECISION);
        funding = ((funding * _getFundingRatio()) / FEE_PRECISION);

        // update collateral index with funding costs
        newCollIndex = (newCollIndex * totalRawColls) / (totalRawColls - funding);
        _updateCollateralIndex(newCollIndex);
      }

      // update interest snapshot
      _updateInterestRate();
    }
  }
```

```
function operate(
    uint256 positionId,
    int256 newRawColl,
    int256 newRawDebt,
    address owner
  ) external onlyPoolManager returns (uint256, int256, int256, uint256)
{
    if (newRawColl == 0 && newRawDebt == 0) revert ErrorNoSupplyAndNoBorrow();
    if (newRawColl != 0 && (newRawColl > -MIN_COLLATERAL && newRawColl < MIN_COLLATERAL)) {
      revert ErrorCollateralTooSmall();
    }
    if (newRawDebt != 0 && (newRawDebt > -MIN_DEBT && newRawDebt < MIN_DEBT)) {
      revert ErrorDebtTooSmall();
    }
    if (newRawDebt > 0 && (_isBorrowPaused() || !IPegKeeper(pegKeeper).isBorrowAllowed())) {
      revert ErrorBorrowPaused();
    }

    OperationMemoryVar memory op;
    // price precision and ratio precision are both 1e18, use min price here
    (, op.price, ) = IPriceOracle(priceOracle).getPrice();
    (op.globalDebt, op.globalColl) = _getDebtAndCollateralShares();
    // @audit Get debt index value
    (op.collIndex, op.debtIndex) = _updateCollAndDebtIndex();
   ......
  }

function redeem(uint256 rawDebts) external onlyPoolManager returns
(uint256 rawColls) {
    if (_isRedeemPaused()) revert ErrorRedeemPaused();

     // @audit Get debt index value
    (uint256 cachedCollIndex, uint256 cachedDebtIndex) =
_updateCollAndDebtIndex();
   ......
}

function rebalance(int16 tick, uint256 maxRawDebts) external
onlyPoolManager returns (RebalanceResult memory result) {

    // @audit Get debt index value
    (uint256 cachedCollIndex, uint256 cachedDebtIndex) =
_updateCollAndDebtIndex();
    (, uint256 price, ) = IPriceOracle(priceOracle).getPrice(); // use
min price
    uint256 node = tickData[tick];
    ......
}

function rebalance(
    uint32 positionId,
    uint256 maxRawDebts
  ) external onlyPoolManager returns (RebalanceResult memory result) {
    _requireOwned(positionId);

    // @audit Get debt index value
    (uint256 cachedCollIndex, uint256 cachedDebtIndex) =
_updateCollAndDebtIndex();
    (, uint256 price, ) = IPriceOracle(priceOracle).getPrice(); // use
min price
    PositionInfo memory position = _getAndUpdatePosition(positionId);
    uint256 positionRawColl = _convertToRawColl(position.colls,
cachedCollIndex, Math.Rounding.Down);
    ......
}
```

Status

The development team confirmed the issue. They explained that under the ideal code execution logic when a position incurs bad debt, liquidators will trigger bad debt allocation upon liquidating the position. Users attempting to evade their debt obligations must redeem assets before liquidation while avoiding triggering the individual debt ratio limit, leaving them with limited options. Additionally, when a user's position debt ratio reaches the `rebalance` threshold, arbitrageurs can assist in repaying the debt, reducing the user's risk of liquidation. If the user's position debt ratio further reaches the liquidation threshold, liquidators can liquidate the position and receive a certain percentage of rewards. Therefore, it is unlikely that the protocol will experience systemic bad debt.

4.3.4 Recommend implementing a validation check within the `_liquidateTick()` function to verify if the calculated `newTick` exceeds the `topTick` value, preventing incorrect maximum tick updates.

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

Location

BasePool.sol#L232-L235

BasePool.sol#L267-L278

TickLogic.sol#L180-L215

Description

The `redeem()` function processes debt redemption by redeeming LSD collateral in descending tick order at specified redemption ratios (excluding ticks where debt exceeds collateral value). The `collShareRedeemed` parameter represents collateral shares to be deducted from the ledger. Due to protocol funding cost deductions, the deducted collateral shares exceed the actual LSD redemption value.

This logic may result in the `_liquidateTick()` function calculating a `newTick` value exceeding the original tick value. Notably, when the original tick equals `topTick`, the newly calculated `newTick` should become the new top tick; however, the current implementation fails to record this value.

Given the uncertain impact of funding cost deductions on the newTick parameter, it is recommended to implement comparison validation between newTick and topTick within the _liquidateTick() function to prevent parameter update errors.

```
function redeem(uint256 rawDebts) external onlyPoolManager returns
(uint256 rawColls) {
        ......

        // redeem at most `maxRedeemRatioPerTick`
        uint256 debtShareToRedeem = (tickDebtShare *
_getMaxRedeemRatioPerTick()) / FEE_PRECISION;
        if (debtShareToRedeem > debtShare) debtShareToRedeem =
debtShare;
        uint256 rawCollRedeemed = (_convertToRawDebt(debtShareToRedeem,
cachedDebtIndex, Math.Rounding.Down) *
          PRECISION) / price;
        uint256 collShareRedeemed =
_convertToCollShares(rawCollRedeemed, cachedCollIndex,
Math.Rounding.Down);

        // @audit The liquidation process simultaneously updates both
tick and top tick values.
        _liquidateTick(tick, collShareRedeemed, debtShareToRedeem,
price);

        debtShare -= debtShareToRedeem;
        rawColls += rawCollRedeemed;

        cachedTotalColls -= collShareRedeemed;
        cachedTotalDebts -= debtShareToRedeem;

        (tick, hasDebt) = tickBitmap.nextDebtPositionWithinOneWord(tick
- 1);
      }
      if (tick == type(int16).min) break;
    }
    _updateDebtAndCollateralShares(cachedTotalDebts, cachedTotalColls);
  }
```

```
// https://github.com/AladdinDAO/fx-protocol-
contracts/blob/486f1cc7eafdc0a55522953272a456c98d6cf532/contracts/core/p
ool/TickLogic.sol#L180-L215
function _liquidateTick(int16 tick, uint256 liquidatedColl, uint256
liquidatedDebt, uint256 price) internal {
    uint32 node = tickData[tick];
    // create new tree node for this tick
    _newTickTreeNode(tick);
    // clear bitmap first, and it will be updated later if needed.
    tickBitmap.flipTick(tick);

    bytes32 value = tickTreeData[node].value;
    bytes32 metadata = tickTreeData[node].metadata;
    uint256 tickColl = value.decodeUint(0, 128);
    uint256 tickDebt = value.decodeUint(128, 128);
    uint256 tickCollAfter = tickColl - liquidatedColl;
    uint256 tickDebtAfter = tickDebt - liquidatedDebt;
    uint256 collRatio = (tickCollAfter * E60) / tickColl;
    uint256 debtRatio = (tickDebtAfter * E60) / tickDebt;

    // update metadata
    metadata = metadata.insertUint(collRatio, 48, 64);
    metadata = metadata.insertUint(debtRatio, 112, 64);

    int256 newTick = type(int256).min;
    if (tickDebtAfter > 0) {
      // partial liquidated, move funds to another tick
      uint32 parentNode;
      // @audit Calculate the new tick.
      (newTick, parentNode) = _addPositionToTick(tickCollAfter,
tickDebtAfter, false);
      metadata = metadata.insertUint(parentNode, 0, 32);
    }
    emit TickMovement(tick, int16(newTick), tickCollAfter,
tickDebtAfter, price);

    // top tick liquidated, update it to new one
    int16 topTick = _getTopTick();
    if (topTick == tick && newTick != int256(tick)) {
      _resetTopTick(topTick);
    }
    tickTreeData[node].metadata = metadata;
  }
```

Status

The development team explained that the debt index does not affect tick calculation. Even with high funding costs, the tick calculation logic remains unchanged, so the presence of funding costs will not result in `newTick > tick`. The formula for tick is `tick = debt shares / coll shares`, which does not involve funding costs.

4.3.5 Vulnerability in the `rebalance()` function due to the absence of maximum debt ratio limitations.

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

Location

BasePool.sol#L250-L284

BasePool.sol#L287-L3442

Description

When LSD collateral prices decline, position debt ratios (leverage) increase due to reduced collateral value. The protocol enables users to repay position debt through the `rebalance()` function, reducing debt ratios and enhancing protocol security. Both `rebalance()` functions only consider minimum debt ratio thresholds, allowing function execution when current position (or tick) debt ratios exceed `rebalanceDebtRatio`. However, they fail to address scenarios where position (tick) debt ratios surpass liquidation thresholds or where debt value exceeds LSD collateral value.

According to current code logic, the following operations are executed:

(1). Calculate the current position (or tick) deposited LSD collateral quantity `positionRawColl`;

(2). Calculate the current position (or tick) required debt repayment in fxUSD tokens `positionRawDebt`;

(3). Invoke the `_getRawDebtToRebalance()` function to calculate the required fxUSD token debt repayment `result.rawDebts` to reduce position (or tick) debt ratio to `rebalanceDebtRatio`;

(4). Calculate corresponding debt share reduction `debtShareToRebalance` for repaying `result.rawDebts` fxUSD tokens;

(5). Calculate redeemable LSD quantity based on current `price`. Additionally, callers receive bonus LSD rewards `result.bonusRawColls` proportional to redemption amount. Due to funding cost deductions, the actual collateral share reduction `collShareToRebalance` exceeds the result for the redeemed LSD quantity.rawColls + result.bonusRawColls`;

(6). Remove corresponding debt shares and collateral shares from the ledger, recalculate tick and adjusted position parameters;

Due to the numerous variables involved in the above steps, quantitative analysis of potential outcomes is challenging. Two potential scenarios are identified:

In Scenario 1, when the position debt ratio exceeds the liquidation threshold and debt value approaches collateral value, users attempting to reduce debt through the `rebalance()` function may trigger unintended consequences. The interaction of `bonusRawColls`, funding cost deductions, and collateral price dynamics can result in near-complete collateral redemption with residual debt, potentially generating abnormally high tick values. This can lead to excessive gas consumption in subsequent `_resetTopTick()` operations or transaction reversion if the collateral is fully depleted while the debt remains.

In Scenario 2, positions exceeding liquidation thresholds with debt values approaching or exceeding collateral value present critical system risks during rebalancing attempts. The calculated `result.rawDebts` may exceed actual position debt, leading to arithmetic underflow during position updates. This implementation lacks explicit error messaging, making it difficult for users to understand transaction failures in such scenarios.

```
function rebalance(int16 tick, uint256 maxRawDebts) external
onlyPoolManager returns (RebalanceResult memory result) {
    (uint256 cachedCollIndex, uint256 cachedDebtIndex) =
_updateCollAndDebtIndex();
    (, uint256 price, ) = IPriceOracle(priceOracle).getPrice(); // use
min price
    uint256 node = tickData[tick];
    bytes32 value = tickTreeData[node].value;
    uint256 tickRawColl = _convertToRawColl(value.decodeUint(0, 128),
cachedCollIndex, Math.Rounding.Down);
    uint256 tickRawDebt = _convertToRawDebt(value.decodeUint(128, 128),
cachedDebtIndex, Math.Rounding.Down);
    (uint256 rebalanceDebtRatio, uint256 rebalanceBonusRatio) =
_getRebalanceRatios();

    // @audit The implementation only requires the current debt ratio to
exceed rebalanceDebtRatio, without imposing restrictions to prevent debt
ratios from surpassing the liquidateDebtRatio threshold.
    // rebalance only debt ratio >= `rebalanceDebtRatio`
    if (tickRawDebt * PRECISION * PRECISION < rebalanceDebtRatio *
tickRawColl * price) {
      revert ErrorRebalanceDebtRatioNotReached();
    }

    // compute debts to rebalance to make debt ratio to
`rebalanceDebtRatio`
    result.rawDebts = _getRawDebtToRebalance(tickRawColl, tickRawDebt,
price, rebalanceDebtRatio, rebalanceBonusRatio);
    if (maxRawDebts < result.rawDebts) result.rawDebts = maxRawDebts;

    uint256 debtShareToRebalance = _convertToDebtShares(result.rawDebts,
cachedDebtIndex, Math.Rounding.Down);
    result.rawColls = (result.rawDebts * PRECISION) / price;
    result.bonusRawColls = (result.rawColls * rebalanceBonusRatio) /
FEE_PRECISION;
    if (result.bonusRawColls > tickRawColl - result.rawColls) {
      result.bonusRawColls = tickRawColl - result.rawColls;
    }
    uint256 collShareToRebalance = _convertToCollShares(
      result.rawColls + result.bonusRawColls,
      cachedCollIndex,
      Math.Rounding.Down
    );

    _liquidateTick(tick, collShareToRebalance, debtShareToRebalance,
price);
    unchecked {
      (uint256 totalDebts, uint256 totalColls) =
_getDebtAndCollateralShares();
      _updateDebtAndCollateralShares(totalDebts - debtShareToRebalance,
totalColls - collShareToRebalance);
    }
  }

  /// @inheritdoc IPool
  function rebalance(
    uint32 positionId,
    uint256 maxRawDebts
  ) external onlyPoolManager returns (RebalanceResult memory result) {
    _requireOwned(positionId);

    (uint256 cachedCollIndex, uint256 cachedDebtIndex) =
_updateCollAndDebtIndex();
```

```
    (, uint256 price, ) = IPriceOracle(priceOracle).getPrice(); // use min price
    PositionInfo memory position = _getAndUpdatePosition(positionId);
    uint256 positionRawColl = _convertToRawColl(position.colls, cachedCollIndex, Math.Rounding.Down);
    uint256 positionRawDebt = _convertToRawDebt(position.debts, cachedDebtIndex, Math.Rounding.Down);
    (uint256 rebalanceDebtRatio, uint256 rebalanceBonusRatio) = _getRebalanceRatios();

    // @audit The implementation only requires the current debt ratio to exceed rebalanceDebtRatio, without imposing restrictions to prevent debt ratios from surpassing the liquidateDebtRatio threshold.
    // rebalance only debt ratio >= `rebalanceDebtRatio`
    if (positionRawDebt * PRECISION * PRECISION < rebalanceDebtRatio * positionRawColl * price) {
      revert ErrorRebalanceDebtRatioNotReached();
    }


    _removePositionFromTick(position);

    // compute debts to rebalance to make debt ratio to `rebalanceDebtRatio`
    result.rawDebts = _getRawDebtToRebalance(
      positionRawColl,
      positionRawDebt,
      price,
      rebalanceDebtRatio,
      rebalanceBonusRatio
    );
    if (maxRawDebts < result.rawDebts) result.rawDebts = maxRawDebts;

    uint256 debtShareToRebalance = _convertToDebtShares(result.rawDebts, cachedDebtIndex, Math.Rounding.Down);
    result.rawColls = (result.rawDebts * PRECISION) / price;
    result.bonusRawColls = (result.rawColls * rebalanceBonusRatio) / FEE_PRECISION;
    if (result.bonusRawColls > positionRawColl - result.rawColls) {
      result.bonusRawColls = positionRawColl - result.rawColls;
    }
    uint256 collShareToRebalance = _convertToCollShares(
      result.rawColls + result.bonusRawColls,
      cachedCollIndex,
      Math.Rounding.Down
    );
    position.debts -= uint104(debtShareToRebalance);
    position.colls -= uint104(collShareToRebalance);

    {
      int256 tick;
      (tick, position.nodeId) = _addPositionToTick(position.colls,
position.debts, false);
      position.tick = int16(tick);
    }
 ......
  }
```

Suggestion

Implement upper-bound debt ratio validation in the `rebalance()` function, preventing function execution when debt ratios exceed liquidation thresholds. Additionally, compare calculated `result.rawDebts` (required fxUSD token repayment to achieve `rebalanceDebtRatio`) against actual position debt, implementing reversion with explicit error messaging when `result.rawDebts` exceeds actual debt quantity.

Status

The development team adopted our suggestion and added a position debt ratio cap check in the `rebalance()` function in commit 324b62, disallowing it from exceeding the liquidation threshold, thereby fixing the issue.

4.3.6 When liquidating a position, special attention must be given to setting the maximum liquidation debt amount `maxRawDebts` to prevent potential liquidation failures.

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

Location

BasePool.sol#L409

Description

When a user's position debt ratio exceeds the liquidation threshold `liquidateDebtRatio`, they can invoke the `liquidate()` function to liquidate their position. If the position debt value exceeds the collateral value, the liquidator must be cautious about the `maxRawDebts` parameter when calling the `liquidate()` function. Incorrectly setting this parameter can result in a liquidation failure.

For example, if `maxRawDebts` is set to `uint256(-1)`, which represents the maximum possible value (effectively aiming to repay the entire debt of the position), the computed value of `result.rawColls` may exceed the actual amount of collateral in the position (`positionRawColl`). This will lead to an issue within the if condition, where a subtraction operation results in an underflow and fails.

```
function liquidate(
    uint256 positionId,
    uint256 maxRawDebts,
    uint256 reservedRawColls
  ) external onlyPoolManager returns (LiquidateResult memory result) {
    _requireOwned(positionId);

    ......

    result.rawColls = (result.rawDebts * PRECISION) / price;
    result.bonusRawColls = (result.rawColls * liquidateBonusRatio) /
FEE_PRECISION;

    // @audit When a position's debt value exceeds its collateral value,
it is essential to carefully manage the `maxRawDebts` parameter to limit
the amount of debt being repaid during liquidation. Failure to do so may
result in the computed value of `result.rawColls` exceeding the
available collateral (`positionRawColl`), which will cause the
transaction to revert due to an invalid state.
    if (result.bonusRawColls > positionRawColl - result.rawColls) {
      uint256 diff = result.bonusRawColls - (positionRawColl -
result.rawColls);
      if (diff < reservedRawColls) result.bonusFromReserve = diff;
      else result.bonusFromReserve = reservedRawColls;
      result.bonusRawColls = positionRawColl - result.rawColls +
result.bonusFromReserve;

      collShareToLiquidate = position.colls;
    } else {
      collShareToLiquidate = _convertToCollShares(
        result.rawColls + result.bonusRawColls,
        cachedCollIndex,
        Math.Rounding.Down
      );
    }
   ......
  }
```

Status

The development team added overflow handling logic for calculation results and adjusted the amounts of collateral and debt during liquidation in commit dc0032c.

4.3.7 Discussion of the `rebalance()` function logic.

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

Location

FxUSDBasePool.sol#L306-L314

FxUSDBasePool.sol#L326-L334

Description

Any user can invoke the `rebalance()` function to help repay debt, reduce the debt ratio, and mitigate systemic bad debt risks. Under the current code logic, the caller can only use the available amount of `fxusd` and `usdc` tokens stored in the `FxUSDBasePool` contract for rebalancing. Afterward, the caller must deposit the actual amount of funds used back into the contract.

Even if the caller has sufficient funds (`fxusd` / `usdc`), if the contract has insufficient balance, the caller will not be able to repay a large portion of the debt in one go and will need to perform multiple rebalances, which leads to higher gas costs. In extreme cases, if the contract has no `fxusd` or `usdc` tokens, the user will be required to first deposit an equivalent amount of `fxusd` / `usdc` tokens into the contract before calling the `rebalance()` function. This introduces significant inconvenience for users attempting to perform liquidation.

Verifying whether this logic aligns with the original design intention is necessary.

```
function rebalance(
    address pool,
    int16 tickId,
    address tokenIn,
    uint256 maxAmount,
    uint256 minCollOut
  ) external onlyValidToken(tokenIn) nonReentrant returns (uint256 tokenUsed, uint256 colls) {
    // @audit The amount of funds used for rebalancing in the current contract is determined based on the available quantity of fxusd / usdc tokens stored in the contract.
    RebalanceMemoryVar memory op = _beforeRebalanceOrLiquidate(tokenIn, maxAmount);
    (op.colls, op.yieldTokenUsed, op.stableTokenUsed) = IPoolManager(poolManager).rebalance(
      pool,
      _msgSender(),
      tickId,
      op.yieldTokenToUse,
      op.stableTokenToUse
    );

    // @audit The caller must deposit the amount of fxusd / usdc tokens they intend to use for rebalancing into the contract.
    tokenUsed = _afterRebalanceOrLiquidate(tokenIn, minCollOut, op);
    colls = op.colls;
  }

function rebalance(
   address pool,
   uint32 positionId,
   address tokenIn,
   uint256 maxAmount,
   uint256 minCollOut
 ) external onlyValidToken(tokenIn) nonReentrant returns (uint256 tokenUsed, uint256 colls) {
   RebalanceMemoryVar memory op = _beforeRebalanceOrLiquidate(tokenIn, maxAmount);
   (op.colls, op.yieldTokenUsed, op.stableTokenUsed) = IPoolManager(poolManager).rebalance(
     pool,
     _msgSender(),
     positionId,
     op.yieldTokenToUse,
     op.stableTokenToUse
   );
   tokenUsed = _afterRebalanceOrLiquidate(tokenIn, minCollOut, op);
   colls = op.colls;
 }
```

Status

The development team confirmed that the logic aligns with the design intent. If the Stability Pool runs out of capital, keepers can participate rebalance/liquidation transactions directly.

4.3.8 Frequent swapping between `fxusd` tokens and `usdc` tokens within the protocol may lead to the erosion of the total asset value held by the protocol.

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

Location

FxUSDBasePool.sol#L359

Description

In the current contract, the price of the `fxusd` token is assumed to be fixed at $1, while the price of the `usdc` token is retrieved from relevant price oracles. Based on this data, the contract executes swap operations between `fxusd` and `usdc`. Due to fluctuations in the price of `usdc`, these frequent swaps may reduce the protocol's total asset value.

For example, in the `arbitrage()` function, the keeper can invoke the function and execute a swap between `fxusd` and `usdc` based on the real-time `usdc` price. Suppose the price of `fxusd` is pegged at $1 in the secondary market, but the `usdc` price is higher than $1, such that: 1 `fxusd` = 0.995 `usdc`. In this case, the `fxusd-usdc` liquidity pool in the Curve protocol would hold more `fxusd` than `usdc` tokens, which aligns with the exchange rate. The keeper can then execute the `arbitrage()` function, swapping 100,000 `fxusd` for 99,950 `usdc`. However, if the price of `usdc` later returns to parity (e.g., 1 `fxusd` = 1.002 `usdc`), the 99,950 `usdc` tokens held by the protocol could only be exchanged for 99,750.5 `fxusd` tokens, resulting in a loss of 249.5 `fxusd` tokens due to price fluctuations. This price volatility of `usdc` leads to a loss in value when executing the swap.

In contrast, if the price of `usdc` is lower than $1 (as per the Chainlink price feed), the `fxusd-usdc` liquidity pool in Curve would hold more `usdc` than `fxusd` tokens, which matches the exchange rate. In this case, the keeper could swap `usdc` for `fxusd`, but the amount of `fxusd` received would be less than the amount of `usdc` provided. When the price of `usdc` returns to $1, the protocol would again suffer a loss due to the price differential.

The core issue here is that the contract assumes the price of fxusd remains fixed at $1, while the price of usdc fluctuates around that value. Frequent swaps based on these fluctuations can erode the total value of the protocol's assets. Each swap may result in buying or selling tokens at unfavorable rates, reducing the overall value held in the system.

A similar value erosion problem can occur in the rebalance() operation.

```
function arbitrage(
    address srcToken,
    uint256 amountIn,
    address receiver,
    bytes calldata data
  ) external onlyValidToken(srcToken) onlyPegKeeper nonReentrant returns
(uint256 amountOut, uint256 bonusOut) {
    address dstToken;
    uint256 expectedOut;
    uint256 cachedTotalYieldToken = totalYieldToken;
    uint256 cachedTotalStableToken = totalStableToken;
    {
      uint256 price = getStableTokenPrice();
      uint256 scaledPrice = price * stableTokenScale;
      if (srcToken == yieldToken) {
        // check if usdc depeg
        if (price < stableDepegPrice) revert ErrorStableTokenDepeg();
        if (amountIn > cachedTotalYieldToken) revert
ErrorSwapExceedBalance();
        dstToken = stableToken;
        unchecked {
          // rounding up
          expectedOut = Math.mulDivUp(amountIn, PRECISION, scaledPrice);
          cachedTotalYieldToken -= amountIn;
          cachedTotalStableToken += expectedOut;
        }
      } else {
        if (amountIn > cachedTotalStableToken) revert
ErrorSwapExceedBalance();
        dstToken = yieldToken;
        unchecked {
          // rounding up
          expectedOut = Math.mulDivUp(amountIn, scaledPrice, PRECISION);
          cachedTotalStableToken -= amountIn;
          cachedTotalYieldToken += expectedOut;
        }
      }
    }
    IERC20(srcToken).safeTransfer(pegKeeper, amountIn);
    uint256 actualOut = IERC20(dstToken).balanceOf(address(this));
    amountOut = IPegKeeper(pegKeeper).onSwap(srcToken, dstToken,
amountIn, data);
    actualOut = IERC20(dstToken).balanceOf(address(this)) - actualOut;
    // check actual fxUSD swapped in case peg keeper is hacked.
    if (amountOut > actualOut) revert ErrorInsufficientOutput();
    // check swapped token has no loss
    if (amountOut < expectedOut) revert ErrorInsufficientArbitrage();

    totalYieldToken = cachedTotalYieldToken;
    totalStableToken = cachedTotalStableToken;
    bonusOut = amountOut - expectedOut;
    if (bonusOut > 0) {
      IERC20(dstToken).safeTransfer(receiver, bonusOut);
    }

    emit Arbitrage(_msgSender(), srcToken, amountIn, amountOut,
bonusOut);
  }
```

Status

The development team confirmed the issue and explained that the `arbitrage()` function is a permissioned function restricted to keepers, making it inaccessible to attackers. The function limits its use to scenarios where the price difference between secondary market USDC and fxUSD exceeds a certain threshold (e.g., 0.3%) to prevent frequent trades. Additionally, to deter arbitrageurs from exploiting the `deposit()` and `withdraw()` functions for repeated arbitrage during exchange rate imbalances, the code introduces a redemption time limit in commit b4efee9, requiring users to hold base pool tokens for a specified duration before redemption .

4.3.9 When using the `onMigrateXstETHPosition()` function to migrate positions, there is a possibility that the minted `fxusd` token debt will be insufficient to repay the borrowed `usdc` token amount, which may result in the migration failing.

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

Location

MigrateFacet.sol#L216

MigrateFacet.sol#L261

MigrateFacet.sol#L224

MigrateFacet.sol#L269

PositionOperateFlashLoanFacet.sol#L188

Description

To facilitate the migration of `xtoken` held by users in fx v2 to fixed positions in fx 2.0, the protocol provides the `migrateXstETHPosition()` and `migrateXfrxETHPosition()` function interfaces for users to interact with. Taking the `migrateXstETHPosition()` function as an example, its internal logic is as follows:

(1). The user deposits `xstETH` tokens from fx v2 and an NFT token for the fx 2.0 position (which will be created if it does not exist).

(2). A flash loan of `borrowAmount` `usdc` tokens is taken from the Balancer protocol. These `usdc` tokens are then swapped for `fTokenAmount` `fxusd` tokens through the relevant Curve pool, converted into an equivalent amount of `ftoken`.

(3). These `xstETH` and `ftoken` are redeemed via the fx v2 protocol, yielding `wstETHAmount` of `wstETH`.

(4). The `wstETHAmount` of `wstETH` is then deposited into the corresponding position in fx 2.0, and at the same time, `fTokenAmount` of `fxusd` tokens are minted as debt.

(5). Finally, the `fxusd` tokens are swapped for `usdc` tokens through the Curve pool to repay the flash loan.

It is important to note that to have enough `usdc` tokens to repay the funds borrowed from the Balancer protocol, the protocol mints `fTokenAmount` `fxusd` tokens when opening (or increasing) the position to cover the debt. Since the swap between `fxusd` and `usdc` tokens via the Curve pool may involve slippage, the minted amount of `fxusd` tokens is increased by 0.1%. However, this may still not be sufficient to repay the borrowed funds. There are several factors to consider, including slippage and fees for the asset swap via Curve pools and fees associated with the flash loan. Therefore, simply increasing the minted `fxusd` by 0.1% may not cover all the fees and borrowed amounts, and the actual amount needed will have to be confirmed when the protocol is live.

Additionally, using borrowAmount as the amount of usdc tokens to be swapped out is not entirely accurate, as the repayment of the flash loan requires paying the borrowing fees as well. Therefore, the amount of usdc tokens to be swapped out needs to be at least borrowAmount + borrowing fees, and not just borrowAmount. This issue is also present in the _swap() function within the PositionOperateFlashLoanFacet contract (link), which is not reiterated here.

The same issue exists in the onMigrateXfrxETHPosition() function.

```
function onMigrateXstETHPosition(
    address pool,
    uint256 positionId,
    uint256 xTokenAmount,
    uint256 borrowAmount,  // @audit usdc tokens from flash loan
    address recipient,
    bytes memory data
  ) external onlySelf {
    uint256 fTokenAmount = (xTokenAmount * IERC20(fstETH).totalSupply()) / IERC20(xstETH).totalSupply();

    ......

    // @audit In practice, in addition to slippage, there are also fees charged by the liquidity pools and the borrowing fees associated with the flash loan. Increasing the minted `fxusd` token amount by 0.1% may not be sufficient to cover all these costs.
    // since we need to swap back to USDC, mint 0.1% more fxUSD to cover slippage.
    fTokenAmount = (fTokenAmount * 1001) / 1000;

    LibRouter.approve(wstETH, poolManager, wstETHAmount);
    positionId = IPoolManager(poolManager).operate(pool, positionId, int256(wstETHAmount), int256(fTokenAmount));
    _checkPositionDebtRatio(pool, positionId, abi.decode(data, (bytes32)));
    IERC721(pool).transferFrom(address(this), recipient, positionId);

    // swap fxUSD to USDC and pay debts
    // @audit Using `borrowAmount` as the minimum amount of `usdc` to be swapped is not entirely accurate, as the flash loan also incurs a borrowing fee. Therefore, the actual amount of `usdc` that needs to be swapped should be at least `borrowAmount + fee`, where `fee` represents the borrowing cost of the flash loan.
    _swapFxUSDToUSDC(IERC20(fxUSD).balanceOf(address(this)),
borrowAmount, data);
  }

function onMigrateXfrxETHPosition(
    address pool,
    uint256 positionId,
    uint256 xTokenAmount,
    uint256 borrowAmount, // @audit usdc tokens from flash loan
    address recipient,
    bytes memory data
  ) external onlySelf {
    uint256 fTokenAmount = (xTokenAmount *
IERC20(ffrxETH).totalSupply()) / IERC20(xfrxETH).totalSupply();

    // swap USDC to fxUSD
    fTokenAmount = _swapUSDCToFxUSD(borrowAmount, fTokenAmount, data);

    // unwrap fxUSD as fToken
    IFxUSD(fxUSD).unwrap(sfrxETH, fTokenAmount, address(this));

    uint256 wstETHAmount;
    {
      // redeem
      wstETHAmount =
IFxMarketV2(sfrxETHMarket).redeemXToken(xTokenAmount, address(this), 0);
      (uint256 baseOut, uint256 bonus) =
IFxMarketV2(sfrxETHMarket).redeemFToken(fTokenAmount, address(this), 0);
      wstETHAmount += baseOut + bonus;
      // swap sfrxETH to wstETH
      wstETHAmount = _swapSfrxETHToWstETH(wstETHAmount, 0, data);
    }

    // @audit In practice, in addition to slippage, there are also fees
charged by the liquidity pools and the borrowing fees associated with
the flash loan. Therefore, increasing the minted `fxusd` token amount by
0.1% may not be sufficient to cover all these costs.
    // since we need to swap back to USDC, mint 0.1% more fxUSD to cover
slippage.
    fTokenAmount = (fTokenAmount * 1001) / 1000;

    LibRouter.approve(wstETH, poolManager, wstETHAmount);
    positionId = IPoolManager(poolManager).operate(pool, positionId,
int256(wstETHAmount), int256(fTokenAmount));
```

```
    _checkPositionDebtRatio(pool, positionId, abi.decode(data,
(bytes32)));
    IERC721(pool).transferFrom(address(this), recipient, positionId);

    // swap fxUSD to USDC and pay debts
    // @audit Using `borrowAmount` as the minimum amount of `usdc` to be
swapped is not entirely accurate, as the flash loan also incurs a
borrowing fee.
    _swapFxUSDToUSDC(IERC20(fxUSD).balanceOf(address(this)),
borrowAmount, data);
  }
```

Status

The development team explained that under the Curve protocol, using an amplification of 0.1% is sufficient as long as the pool remains pegged. Currently, the Balancer protocol does not charge flash loan fees, which is not accounted for in the code logic.

4.3.10 The remaining funds in the contract should be sent to the caller.

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

Location

PositionOperateFlashLoanFacet.sol#L117

PositionOperateFlashLoanFacet.sol#L155

Description

The user can use the openOrAddPositionFlashLoan() function in this contract to convert their held lsd tokens into leveraged positions without needing to hold fxusd tokens. Since the funds, including the flash loan, are provided entirely by the caller, the remaining funds in the contract should belong to the caller. On the other hand, the remaining lsd tokens in the contract are likely the funds left over after the caller repaid the flash loan. These tokens were exchanged for fxusd tokens by the caller, and thus should also belong to the caller. If the current code has been modified so that the remaining lsd tokens are sent to the revenuePool instead of being returned to the caller, this could raise concerns. It is essential to clarify why the contract now sends the remaining lsd tokens to the revenuePool.

```
function openOrAddPositionFlashLoan(
    LibRouter.ConvertInParams memory params,
    address pool,
    uint256 positionId,
    uint256 borrowAmount,
    bytes calldata data
  ) external payable onFlashLoan {
    uint256 amountIn = LibRouter.transferInAndConvert(params,
IPool(pool).collateralToken()) + borrowAmount;

    address[] memory tokens = new address[](1);
    uint256[] memory amounts = new uint256[](1);
    tokens[0] = IPool(pool).collateralToken();
    amounts[0] = borrowAmount;
    IBalancerVault(balancer).flashLoan(
      address(this),
      tokens,
      amounts,
      abi.encodeCall(
        PositionOperateFlashLoanFacet.onOpenOrAddPositionFlashLoan,
        (pool, positionId, amountIn, borrowAmount, msg.sender, data)
      )
    );

    // refund collateral token to caller
    // @audit The remaining funds in the contract may be the funds left
over by the caller after repaying the flash loan, which belong to the
caller. If these funds are instead sent to the revenuePool address, it
would not align with the original intent of returning the funds to the
caller.
    LibRouter.refundERC20(IPool(pool).collateralToken(), revenuePool);
  }

function closeOrRemovePositionFlashLoan(
  LibRouter.ConvertOutParams memory params,
  address pool,
  uint256 positionId,
  uint256 amountOut,
  uint256 borrowAmount,
  bytes calldata data
) external onFlashLoan {
  address collateralToken = IPool(pool).collateralToken();

  address[] memory tokens = new address[](1);
  uint256[] memory amounts = new uint256[](1);
  tokens[0] = collateralToken;
  amounts[0] = borrowAmount;
  IBalancerVault(balancer).flashLoan(
    address(this),
    tokens,
    amounts,
    abi.encodeCall(
      PositionOperateFlashLoanFacet.onCloseOrRemovePositionFlashLoan,
      (pool, positionId, amountOut, borrowAmount, msg.sender, data)
    )
  );

  // convert collateral token to other token
  amountOut = IERC20(collateralToken).balanceOf(address(this));
  LibRouter.convertAndTransferOut(params, collateralToken, amountOut,
msg.sender);

  // refund rest fxUSD and leveraged token
  // @audit The remaining funds in the contract may be the funds left
over after the caller exchanges the flash loan lsd tokens for fxusd
tokens, which should belong to the caller.
  LibRouter.refundERC20(fxUSD, revenuePool);
}
```

Status

The development team confirmed the issue and explained that this portion of the funds is allocated as incentives to stability pool mostly.

4.3.11 Bypassing the `flashLoanContext` check in the `FlashLoanCallbackFacet` contract could allow an attacker to exploit the contract logic.

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

Location

[MigrateFacet.sol#L135-L143]

[MigrateFacet.sol#L171-L179]

[PositionOperateFlashLoanFacet.sol#L106-L114]

[PositionOperateFlashLoanFacet.sol#L140-L148]

Description

The `openOrAddPositionFlashLoan()` function in the `PositionOperateFlashLoanFacet` contract interacts with the external Balancer contract's `flashLoan()` interface to execute flash loan operations. The `onFlashLoan()` modifier is employed to manage the `flashLoanContext` state of the Router contract, setting the state to `HAS_FLASH_LOAN` upon entering the function and reverting it to `NOT_FLASH_LOAN` upon exiting.

In contrast, the `receiveFlashLoan()` function within the `FlashLoanCallbackFacet` contract serves as a callback for the Balancer Vault contract. This function enforces that it can only be invoked by the Balancer Vault contract and requires the `flashLoanContext` state to be `HAS_FLASH_LOAN`. These context checks are designed to control the contract's call chain and parameter passing, mitigating risks of unauthorized function misuse.

Notably, the function includes a customizable operation allowing the contract to be called via `address(this).call(userData)`. Ideally, due to the constraints enforced by the aforementioned context checks, `userData` cannot be arbitrarily controlled by users. It is restricted to calling only specific functions with tightly regulated parameters.

For example, one scenario involves invoking the `onOpenOrAddPositionFlashLoan()` function through `userData`. This function is protected by the `onlySelf()` modifier, ensuring it can only be called internally by the contract itself. In principle, this design restricts invocations to legitimate call chains, as described earlier.

Now, let’s consider an attacker bypassing the `flashLoanContext` checks. Specifically, the attacker manipulates the execution phase of the `openOrAddPositionFlashLoan()` function to redirect control to a malicious contract. A key observation is the existence of the `params.tokenIn` parameter in the `transferInAndConvert()` function within `LibRouter`, which is user-controlled and triggers external calls. This parameter can be exploited to point to a malicious contract.

In this malicious contract, the attacker calls the Balancer `flashLoan(recipient, tokens, amounts, userData)` interface, setting the `recipient` to this Router contract, namely the `FlashLoanCallbackFacet` contract. As a result, the attacker successfully gains access to the `receiveFlashLoan()` function and passes the prior context validation. At this point, `userData` is entirely under the attacker’s control.

If the attacker invokes the `onOpenOrAddPositionFlashLoan()` function through `userData`, they can successfully bypass the `onlySelf()` validation. By setting the `recipient` to any arbitrary victim, the attacker could forcefully transfer the victim's Pool ERC721 Token via `transferFrom` to the Router contract for further processing.

This compromises the intended call chain. In theory, the attacker could perform arbitrary operations under the guise of the Router contract.

The above illustrates one potential attack scenario. In practice, other functions protected by `onFlashLoan()` and `onlySelf()` might also pose risks, though the complexity of exploiting them may vary.

```
// located in PositionOperateFlashLoanFacet.sol
modifier onlySelf() {
  if (msg.sender != address(this)) revert ErrorNotFromSelf();
  _;
}

modifier onFlashLoan() {
  LibRouter.RouterStorage storage $ = LibRouter.routerStorage();
  $.flashLoanContext = LibRouter.HAS_FLASH_LOAN;
  _;
  $.flashLoanContext = LibRouter.NOT_FLASH_LOAN;
}

function openOrAddPositionFlashLoan(
  LibRouter.ConvertInParams memory params,
  address pool,
  uint256 positionId,
  uint256 borrowAmount,
  bytes calldata data
) external payable onFlashLoan {
  uint256 amountIn = LibRouter.transferInAndConvert(params,
IPool(pool).collateralToken()) + borrowAmount;

  address[] memory tokens = new address[](1);
  uint256[] memory amounts = new uint256[](1);
  tokens[0] = IPool(pool).collateralToken();
  amounts[0] = borrowAmount;
  IBalancerVault(balancer).flashLoan(
    address(this),
    tokens,
    amounts,
    abi.encodeCall(
      PositionOperateFlashLoanFacet.onOpenOrAddPositionFlashLoan,
      (pool, positionId, amountIn, borrowAmount, msg.sender, data)
    )
  );
  // refund collateral token to caller
  LibRouter.refundERC20(IPool(pool).collateralToken(), revenuePool);
}

function onOpenOrAddPositionFlashLoan(
  address pool,
  uint256 position,
  uint256 amount,
  uint256 repayAmount,
  address recipient,
  bytes memory data
) external onlySelf {
  (bytes32 miscData, uint256 fxUSDAmount, uint256 swapEncoding,
uint256[] memory swapRoutes) = abi.decode(
    data,
    (bytes32, uint256, uint256, uint256[])
  );

  // open or add collateral to position
  if (position != 0) {
    IERC721(pool).transferFrom(recipient, address(this), position);
  }
  LibRouter.approve(IPool(pool).collateralToken(), poolManager, amount);
  position = IPoolManager(poolManager).operate(pool, position,
int256(amount), int256(fxUSDAmount));
  _checkPositionDebtRatio(pool, position, miscData);
  IERC721(pool).transferFrom(address(this), recipient, position);

  // swap fxUSD to collateral token
  _swap(fxUSD, fxUSDAmount, repayAmount, swapEncoding, swapRoutes);
}

// located in FlashLoanCallbackFacet.sol
function receiveFlashLoan(
  address[] memory tokens,
  uint256[] memory amounts,
  uint256[] memory feeAmounts,
  bytes memory userData
) external {
  if (msg.sender != balancer) revert ErrorNotFromBalancer();

  // make sure call invoked by router
  LibRouter.RouterStorage storage $ = LibRouter.routerStorage();
  if ($.flashLoanContext != LibRouter.HAS_FLASH_LOAN) revert
ErrorNotFromRouterFlashLoan();
```

```
  (bool success, ) = address(this).call(userData);
  // below lines will propagate inner error up
  if (!success) {
    // solhint-disable-next-line no-inline-assembly
    assembly {
      let ptr := mload(0x40)
      let size := returndatasize()
      returndatacopy(ptr, 0, size)
      revert(ptr, size)
    }
  }

  for (uint256 i = 0; i < tokens.length; i++) {
    IERC20(tokens[i]).safeTransfer(msg.sender, amounts[i] +
feeAmounts[i]);
  }
}


// located in LibRouter.sol
function transferInAndConvert(ConvertInParams memory params, address
tokenOut) internal returns (uint256 amountOut) {
  RouterStorage storage $ = routerStorage();
  if (!$.approvedTargets.contains(params.target)) {
    revert ErrorTargetNotApproved();
  }

  transferTokenIn(params.tokenIn, address(this), params.amount);

  amountOut = IERC20(tokenOut).balanceOf(address(this));
  if (params.tokenIn == tokenOut) return amountOut;

  bool _success;
  if (params.tokenIn == address(0)) {
    (_success, ) = params.target.call{ value: params.amount }
(params.data);
  } else {
    address _spender = $.spenders[params.target];
    if (_spender == address(0)) _spender = params.target;

    approve(params.tokenIn, _spender, params.amount);
    (_success, ) = params.target.call(params.data);
  }
```

```
    // below lines will propagate inner error up
    if (!_success) {
      // solhint-disable-next-line no-inline-assembly
      assembly {
        let ptr := mload(0x40)
        let size := returndatasize()
        returndatacopy(ptr, 0, size)
        revert(ptr, size)
      }
    }

    amountOut = IERC20(tokenOut).balanceOf(address(this)) - amountOut;
  }
```

Status

The development team confirmed the issue and resolved it in commit 240d865.

5. Conclusion

After auditing and analyzing the f(x) Protocol 2.0, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above.

Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability, and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company, or investment.

APPENDIX

Vulnerability/Risk Level Classification

Level	Description
High	Severely damage the contract's integrity and allow attackers to steal Ethers and tokens, or lock assets inside the contract.
Medium	Damage contract's security under given conditions and cause impairment of benefit for stakeholders.
Low	Cause no actual impairment to contract.
Info	Relevant to practice or rationality of the smart contract, could possibly bring risks.

SECBIT Lab is devoted to constructing a common-consensus, reliable, and ordered blockchain economic entity.

🌎 https://secbit.io

✉ audit@secbit.io

🐦 @secbit_io