Security Audit Report

The f(x) Protocol Updates:

Batch Position Management

WBTC Oracle Implementation

Morpho Flashloan Integration

SECBIT

March 17, 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 upgraded version of the f(x) protocol has implemented the following three improvements:

- To mitigate the impact of extreme market conditions on collateral value fluctuations, the protocol has added batch position rebalancing and batch liquidation functions.
- The protocol has introduced WBTC price oracle, allowing users to open positions using WBTC as collateral.
- The protocol has integrated Morpho's flash loan functionality, providing users with more options for opening, adjusting, and closing positions.

SECBIT Labs conducted an audit from February 20 to March 14, 2025, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the upgraded version of f(x) protocol contracts have 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 Discussion on batch position adjustment logic via the `rebalance` function.	Info	Discussed
Design & Implementation	4.3.2 Discussion on the code logic for checking minimum debt during batch liquidation.	Info	Discussed
Design & Implementation	4.3.3 Under certain conditions, the bitmap data used to mark whether a specific tick has an active user position may become incorrect.	Medium	Fixed
Design & Implementation	4.3.4 Using the Chainlink WBTC/BTC price feed may be subject to arbitrage.	Medium	Fixed
Design & Implementation	4.3.5 Discussion on the function `_checkPositionDebtRatio()`.	Info	Discussed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the upgraded version of the f(x) protocol are shown below:

- Smart contract code
    - initial review commit
        - batch rebalance and liquidation: *1b46767*
        - WBTC price oracle: *d51b7ae*
        - use morpho to flashloan: *7d78fc7*
    - final review commit
        - batch rebalance and liquidation: *6498785*
        - WBTC price oracle: *a0c258e*
        - use morpho to flashloan: *7d78fc7*

2.2 Contract List

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

Name	Lines	Description
BasePool.sol	455	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.
TickLogic.sol	157	The supplementary module of the protocol calculates corresponding tick values based on position debt ratios, facilitating position liquidation processes through tick-based execution mechanisms.
FxUSDBasePool.sol	461	Users can deposit fxUSD and USDC tokens into the protocol contract, where these assets are utilized for position leverage adjustment and protocol risk mitigation.
PoolManager.sol	466	The peripheral contract provides direct interfaces for users to execute position operations, including position opening, closing, adjustment, and liquidation procedures.
WBTCPriceOracle.sol	45	This contract provides on-chain price data for WBTC to the f(x) protocol.
BTCDerivativeOracleBase.sol	78	Abstract contract that assists in providing WBTC price data.
MorphoFlashLoanCallbackFacet.sol	30	Implemented the callback function for the Morpho protocol, invoking operations such as opening, adjusting, and closing positions via the `call` instruction.
MorphoFlashLoanFacetBase.sol	27	Abstract contract providing functions to interact with Morpho protocol's flash loan interface.
PositionOperateFlashLoanFacetV2.sol	147	Auxiliary contract of the FX protocol, utilizing Morpho protocol's flash loan for opening, adjusting, and closing positions.

Notice: This audit specifically focuses on the modified portions of the BasePool.sol, TickLogic.sol, FxUSDBasePool.sol, and PoolManager.sol contracts.

3. Contract Analysis

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

3.1 Role Classification

Two key roles in the upgraded version of f(x) protocol are the Governance Account and Common Account.

- Governance Account
    - Description

Contract Administrator

- Authority
    - Update fees ratio
    - Transfer ownership
    - Update market and incentive configurations
    - 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 FX protocol

- 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 version of the protocol introduces batch rebalancing and batch liquidation functions, allowing for quick risk position adjustments and reducing user losses. Meanwhile, building upon the existing support for the Balancer protocol, the protocol now enables using Morpho protocol's flash loan services for opening, adjusting, and closing positions, offering users more options. Additionally, the protocol has integrated a WBTC price oracle, providing users with more collateral options.

We can divide the critical functions of the contract into three parts:

PoolManager and BasePool

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:

- `rebalance(uint256)`

When the collateral price drops, causing a user's debt ratio to become too high, any user can call this function to repay the user's debt and reduce their position leverage. This function will batch-adjust user positions based on their debt ratio, prioritizing those with the highest debt ratios first.

- `liquidate(uint256,uint256)`

When position debt ratios exceed liquidation thresholds, users can invoke this function to execute position liquidation and receive liquidation incentives. This function will batch liquidate user positions based on their debt ratio, prioritizing those with the highest debt ratios first.

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:

- `instantRedeem()`

This function allows users to instantly redeem fxUSD and USDC by paying a certain percentage of fees.

MorphoFlashLoanCallbackFacet、MorphoFlashLoanFacetBase and PositionOperateFlashLoanFacetV2

These three contracts provide flash loan services from the Morpho protocol, enabling users to open positions and perform related operations efficiently. The main functions in these contracts are as follows:

- `openOrAddPositionFlashLoanV2()`

Users can use this function to leverage Morpho protocol's flash loan services to open or increase positions.

- `closeOrRemovePositionFlashLoanV2()`

Users can use this function to leverage Morpho protocol's flash loan services to close a position or remove collateral from the position.

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 Discussion on batch position adjustment logic via the `rebalance` function.

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

Location

BasePool.sol#L312-L361

Description

The protocol has introduced the `rebalance()` function, which rebalances user positions in descending order based on tick values. Specifically, the protocol will iterate from the highest tick (representing the highest debt ratio) and perform the following operations for each valid tick:

1). Retrieve the position data for the tick, including collateral shares, debt shares, and the raw values of collateral and debt.

2). Skip the tick which meets one of the following conditions:

- Already at the liquidation threshold (handled by a separate liquidation function).
- Debt is below the minimum debt limit (to prevent calculation errors).
- Positions that have not yet reached the rebalance threshold (subsequent ticks will also not meet the conditions).

3). Perform the rebalance operation for qualified ticks:

- Calculate and process a portion of the debt (`rawDebts`).
- Calculate and redeem the corresponding collateral (`rawColls`).
- Calculate and distribute rewards (`bonusRawColls`).

This function currently only allows execution from the highest tick, skipping invalid ticks while processing valid ones sequentially in descending order. However, this design presents two inefficiencies:

1). If some positions are already awaiting rebalancing, the function must start from the top tick and check each tick one by one, skipping those that do not meet rebalance conditions. This results in unnecessary gas consumption.

2). If many positions require rebalance and are densely distributed across different tick values, the current implementation processes them sequentially from the highest tick downward. This prevents multiple bots from handling different ticks in parallel. This could delay the rebalance process in volatile market conditions and lead to unnecessary liquidations.

To address these issues, it is recommended that the `rebalance()` function include an optional starting tick parameter, allowing users to specify the tick from which rebalance operations should begin.

```
function rebalance(uint256 maxRawDebts) external onlyPoolManager returns
(RebalanceResult memory result) {
    RebalanceVars memory vars;
    vars.maxRawDebts = maxRawDebts;
    (vars.rebalanceDebtRatio, vars.rebalanceBonusRatio) =
_getRebalanceRatios();
    (, vars.price, ) = IPriceOracle(priceOracle).getPrice();
    (vars.collIndex, vars.debtIndex) = _updateCollAndDebtIndex();
    (vars.totalDebtShares, vars.totalCollShares) =
_getDebtAndCollateralShares();
    (uint256 liquidateDebtRatio, ) = _getLiquidateRatios();

    int16 tick = _getTopTick();
    bool hasDebt = true;
    while (vars.maxRawDebts > 0) {
      if (!hasDebt) {
        (tick, hasDebt) = tickBitmap.nextDebtPositionWithinOneWord(tick
- 1);
      } else {
        ......
    }

    _updateDebtAndCollateralShares(vars.totalDebtShares,
vars.totalCollShares);
  }
```

Status

The development team has acknowledged this issue and explained that the previous implementation already provides a rebalance(int16, uint256) function, which allows position rebalance for a specific tick.

4.3.2 Discussion on the code logic for checking minimum debt during batch liquidation.

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

Location

BasePool.sol#L406-L410

BasePool.sol#L339-L343

Description

The current logic of the `liquidate()` function processes liquidation in a sequential manner, starting from the highest tick (representing the highest debt ratio). The function iterates through each valid tick and performs the following steps:

1). Retrieve the position data for the tick, including collateral shares, debt shares, and the raw values of collateral and debt.

2). Check liquidation conditions:

- If the debt ratio of the tick does not exceed the liquidation threshold, all subsequent ticks will also fail to meet the liquidation condition, the function should break the loop and finish the liquidation process.
- Skip positions with debt below the minimum debt threshold to avoid calculation errors.

3). Perform liquidation for valid ticks:

- Calculate and settle debt (`rawDebts`).
- Redeem corresponding collateral (`rawColls`).
- Compute liquidation bonus (`bonusRawColls`).
- Determine additional subsidies from the reserve pool (`bonusFromReserve`).

In the current implementation, if a tick does not meet the liquidation threshold, the function should execute a `break` statement to exit the `while` loop. However, the minimum debt check is placed inside the liquidation condition check, making it redundant in cases where the tick fails the liquidation condition. This could result in unnecessary tick calculations and wasted gas.

Instead, the minimum debt check should be moved after the liquidation condition check. This way, if a tick is already ineligible for liquidation, the function will exit early, avoiding unnecessary computations. The same optimization should also be applied to the batch `rebalance()` function to improve efficiency and reduce gas consumption.

```
function liquidate(
    uint256 maxRawDebts,
    uint256 reservedRawColls
  ) external onlyPoolManager returns (LiquidateResult memory result) {
    LiquidateVars memory vars;
    vars.maxRawDebts = maxRawDebts;
    vars.reservedRawColls = reservedRawColls;
    (vars.liquidateDebtRatio, vars.liquidateBonusRatio) =
_getLiquidateRatios();
```

```
    (, vars.price, ) = IPriceOracle(priceOracle).getPrice();
    (vars.collIndex, vars.debtIndex) = _updateCollAndDebtIndex();
    (vars.totalDebtShares, vars.totalCollShares) =
_getDebtAndCollateralShares();

    int16 tick = _getTopTick();
    bool hasDebt = true;
    while (vars.maxRawDebts > 0) {
      if (!hasDebt) {
        (tick, hasDebt) = tickBitmap.nextDebtPositionWithinOneWord(tick
- 1);
      } else {
        (vars.tickCollShares, vars.tickDebtShares, vars.tickRawColls,
vars.tickRawDebts) = _getTickRawCollAndDebts(
          tick,
          vars.collIndex,
          vars.debtIndex
        );

        // @audit when this condition is met, it indicates that the
current tick and all subsequent (lower) ticks do not satisfy the
liquidation conditions. Therefore, the liquidation process should stop,
and no further ticks should be processed.
        // no more liquidatable tick: coll * price * liquidateDebtRatio
> debts
        if (vars.tickRawColls * vars.price * vars.liquidateDebtRatio >
vars.tickRawDebts * PRECISION * PRECISION) {
          // skip dust, since the results might be wrong
          if (vars.tickRawDebts < uint256(MIN_DEBT)) {
            hasDebt = false;
            tick = tick;
            continue;
          }
          break;
        }
        // rebalance this tick
        (uint256 rawDebts, uint256 rawColls, uint256 bonusRawColls,
uint256 bonusFromReserve) = _liquidateTick(
          tick,
          vars
        );
        result.rawDebts += rawDebts;
        result.rawColls += rawColls;
        result.bonusRawColls += bonusRawColls;
        result.bonusFromReserve += bonusFromReserve;
```

```
            // goto next tick
            (tick, hasDebt) = tickBitmap.nextDebtPositionWithinOneWord(tick
- 1);
        }
        if (tick == type(int16).min) break;
    }

    _updateDebtAndCollateralShares(vars.totalDebtShares,
vars.totalCollShares);
    _updateDebtIndex(vars.debtIndex);
}
```

Status

The development team explained that the current code is designed to handle both high-debt-ratio positions and small-debt positions during liquidation. These mini-positions are not processed elsewhere in the system, and if left unattended, they could eventually lead to hidden system bad debt. Although the amount of such bad debt is minimal and does not pose a significant threat to system security, addressing them during liquidation helps maintain overall protocol integrity.

4.3.3 Under certain conditions, the bitmap data used to mark whether a specific tick has an active user position may become incorrect.

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

Location

BasePool.sol#L602

Description

A liquidator can use the `liquidate()` function to batch liquidate positions where the debt ratio is greater than or equal to the liquidation threshold. To illustrate the issue, assume that a user position exclusively occupies a single tick, called `tick0`. When `tick0` reaches the liquidation threshold, the liquidator calls `liquidate()`, fully repaying the user's debt while leaving behind a small amount of collateral. The execution logic is as follows:

1). calling `_liquidateTick()` to liquidate the debt in `tick0`

The function _liquidateTick() within the TickLogic contract performs the following operations:

- Calls _newTickTreeNode(), updating the tick's node and initializing tickTreeData[node].metadata. After this, queries to tickData[tick] will return the latest node value.
- Calls tickBitmap.flipTick(tick), which toggles the bit corresponding to tick0 in the bitmap. Since flipTick() inverts the bit, if the bit is 1, it is changed to 0, and vice versa. The purpose of the bitmap is to facilitate checking whether a given tick contains any active user positions. If a tick has an active position, its corresponding bit in the bitmap is set to 1. Under the given assumption, calling this function sets tick0's bitmap bit to 0, indicating no active position in tick0.
- Updates the post-liquidation collateral and debt data in the original node. Under the given assumption, the node has no remaining debt after liquidation but still contains a small amount of collateral. These values remain stored in the old node and can be retrieved using tickTreeData[node].metadata.

At this point, tick0's bitmap bit is set to 0, indicating that it no longer contains any assets.

2). The user calls operate() to close the position and withdraw remaining collateral

- Calls _getAndUpdatePosition() to update the user's position, reflecting zero debt and a small amount of remaining collateral.
- Calls _removePositionFromTick() to remove the user's position from tick0. Since the user was the only one occupying tick0, and the tick no longer has any debt following batch liquidation, _removePositionFromTick() flips the corresponding bitmap bit for tick0 back to 1.

At this moment, tick0 does not contain an active position, yet its bitmap bit is incorrectly set to 1.

3). user calls operate() again to open a new position in tick0

- Calls _addPositionToTick() to add a new position in tick0.
- Since newDebts == debts, the function tickBitmap.flipTick(tick0) is called again, flipping the bit back to 0.

After these steps, tick0 contains an active position, but its bitmap bit is incorrectly set to 0.

Since rebalance() relies on the bitmap to locate active ticks, tick0 will not be rebalanced. If tick0 enters liquidation conditions but is not found in the bitmap, the system will be unable to liquidate it, potentially leading to bad debt.

```
function _liquidateTick(
  int16 tick,
  LiquidateVars memory vars
) internal returns (uint256 rawDebts, uint256 rawColls, uint256
bonusRawColls, uint256 bonusFromReserve) {
  uint256 virtualTickRawColls = vars.tickRawColls +
vars.reservedRawColls;
  rawDebts = vars.tickRawDebts;
  if (rawDebts > vars.maxRawDebts) rawDebts = vars.maxRawDebts;
  rawColls = (rawDebts * PRECISION) / vars.price;
  uint256 debtShares;
  uint256 collShares;
  if (rawDebts == vars.tickRawDebts) {
    // full liquidation
    debtShares = vars.tickDebtShares;
  } else {
    // partial liquidation
    debtShares = _convertToDebtShares(rawDebts, vars.debtIndex,
Math.Rounding.Down);
  }
  if (virtualTickRawColls <= rawColls) {
    // even reserve funds cannot cover bad debts, no bonus and will
trigger bad debt redistribution
    rawColls = virtualTickRawColls;
    bonusFromReserve = vars.reservedRawColls;
    rawDebts = (virtualTickRawColls * vars.price) / PRECISION;
    debtShares = _convertToDebtShares(rawDebts, vars.debtIndex,
Math.Rounding.Down);
    collShares = vars.tickCollShares;
  } else {
    // Bonus is from colls in tick, if it is not enough will use reserve
funds
    bonusRawColls = (rawColls * vars.liquidateBonusRatio) /
FEE_PRECISION;
    uint256 rawCollWithBonus = bonusRawColls + rawColls;
    if (rawCollWithBonus > virtualTickRawColls) {
      rawCollWithBonus = virtualTickRawColls;
      bonusRawColls = rawCollWithBonus - rawColls;
    }
    if (rawCollWithBonus >= vars.tickRawColls) {
      bonusFromReserve = rawCollWithBonus - vars.tickRawColls;
      collShares = vars.tickCollShares;
    } else {
      collShares = _convertToCollShares(rawCollWithBonus,
vars.collIndex, Math.Rounding.Down);
    }
  }

  vars.reservedRawColls -= bonusFromReserve;
  if (collShares == vars.tickCollShares && debtShares <
vars.tickDebtShares) {
    // trigger bad debt redistribution
    uint256 rawBadDebt = _convertToRawDebt(vars.tickDebtShares -
debtShares, vars.debtIndex, Math.Rounding.Down);
    debtShares = vars.tickDebtShares;
    vars.totalCollShares -= collShares;
    vars.totalDebtShares -= debtShares;
    vars.debtIndex += (rawBadDebt * E96) / vars.totalDebtShares;
  } else {
    vars.totalCollShares -= collShares;
    vars.totalDebtShares -= debtShares;
  }
  vars.maxRawDebts -= rawDebts;
  // @audit note the function logic under the assumed conditions
  _liquidateTick(tick, collShares, debtShares, vars.price);
}
```

```
// @audit located in TickLogic, use original logic
function _liquidateTick(int16 tick, uint256 liquidatedColl, uint256
liquidatedDebt, uint256 price) internal {
  uint48 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);

  // @audit read the original node data
  bytes32 value = tickTreeData[node].value;
  bytes32 metadata = tickTreeData[node].metadata;

  // @audit get the coll share and debt share in the tick
  uint256 tickColl = value.decodeUint(COLL_SHARE_OFFSET, 128);
  uint256 tickDebt = value.decodeUint(DEBT_SHARE_OFFSET, 128);

  // @audit update
  uint256 tickCollAfter = tickColl - liquidatedColl;
  uint256 tickDebtAfter = tickDebt - liquidatedDebt;
  uint256 collRatio = (tickCollAfter * E60) / tickColl;
  uint256 debtRatio = (tickDebtAfter * E60) / tickDebt;

  // update metadata
  metadata = metadata.insertUint(collRatio, COLL_RATIO_OFFSET, 64);
  metadata = metadata.insertUint(debtRatio, DEBT_RATIO_OFFSET, 64);

  int256 newTick = type(int256).min;

  if (tickDebtAfter > 0) {
    // partial liquidated, move funds to another tick
    uint48 parentNode;
    (newTick, parentNode) = _addPositionToTick(tickCollAfter,
tickDebtAfter, false);
    metadata = metadata.insertUint(parentNode, PARENT_OFFSET, 48);
  }
  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;
}
```

```
/// @dev Internal function to remove position from tick.
/// @param position The position struct to remove.
function _removePositionFromTick(PositionInfo memory position) internal
{
  // @audit the position has been liquidated
  if (position.nodeId == 0) return;

  bytes32 value = tickTreeData[position.nodeId].value;
  // @audit the original liquidated positions will be merged under the
new node id, minus here first
  uint256 newColls = value.decodeUint(0, 128) - position.colls;
  uint256 newDebts = value.decodeUint(128, 128) - position.debts;

  // @audit remove the corresponding position under nodeid first (the
nodeid of the initial position)
  value = value.insertUint(newColls, 0, 128);
  value = value.insertUint(newDebts, 128, 128);
  // @audit update data
  tickTreeData[position.nodeId].value = value;

  if (newDebts == 0) {
    int16 tick =
int16(tickTreeData[position.nodeId].metadata.decodeInt(32, 16));
    // @audit clear bitmap
    tickBitmap.flipTick(tick);

    // top tick gone, update it to new one
    int16 topTick = _getTopTick();
    if (topTick == tick) {
      _resetTopTick(topTick);
    }
  }
}
```

```
function _addPositionToTick(
  uint256 colls,
  uint256 debts,
  bool checkDebts
) internal returns (int256 tick, uint32 node) {
  if (debts > 0) {
    if (checkDebts && int256(debts) < MIN_DEBT) {
      revert ErrorDebtTooSmall();
    }

    // @audit calculate tick
    tick = _getTick(colls, debts);

    node = _getOrCreateTickNode(tick);

    bytes32 value = tickTreeData[node].value;
    uint256 newColls = value.decodeUint(0, 128) + colls;
    uint256 newDebts = value.decodeUint(128, 128) + debts;
    value = value.insertUint(newColls, 0, 128);
    value = value.insertUint(newDebts, 128, 128);
    tickTreeData[node].value = value;

    // @audit first-created tick, or no-debt tick
    if (newDebts == debts) {
      tickBitmap.flipTick(int16(tick));
    }
```

```
      // update top tick
      if (tick > _getTopTick()) {
        _updateTopTick(int16(tick));
      }
    }
  }
```

Status

The development team fixed this issue in commit 772322e. If liquidation results in a user position with no outstanding debt, the corresponding tick bit will no longer be flipped when removing the user position.

4.3.4 Using the Chainlink WBTC/BTC price feed may be subject to arbitrage.

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

Description

When the LSD asset is WBTC, the price oracle calculates the anchor price using the following method:

WBTC-USDC = Chainlink WBTC-BTC spot price × Chainlink BTC-USDC spot price

The Chainlink WBTC-BTC spot price source is: https://data.chain.link/feeds/ethereum/mainnet/wbtc-btc. This price updates only if the WBTC-BTC price fluctuates by more than 2% or if the update interval exceeds 24 hours. Given the current high price of BTC, assuming $100,000 per BTC, a 2% fluctuation amounts to $2,000. This means that as long as the WBTC-BTC price fluctuates within ±$2,000, the oracle price remains unchanged, creating a potential arbitrage opportunity for attackers.

For example, based on historical WBTC-BTC data from January 7, 2025, the Chainlink oracle reported an exchange rate of 0.996214. On January 8, 2025, this rate increased to 0.998788. If we assume the oracle updates the price at 0.996214 (due to the price movement being less than 2%, triggering the 24-hour update rule), then even though the actual market rate has risen to 0.998788, the oracle still returns 0.996214. This results in the oracle price lagging behind the actual market price.

Additionally, the current protocol's oracle logic fetches the latest spot price data from designated protocols. If the highest price among these sources deviates from the anchor price by more than `maxPriceDeviation` (currently set at 1%), the protocol will use the anchor price as the effective maximum price. An attacker could manipulate the spot price in one of the selected protocols to create a price deviation exceeding `maxPriceDeviation`, forcing the f(x) protocol to adopt the anchor price as the maximum price. This attack is feasible since the cost of manipulating a pool's price by more than 1% is likely low.

At this point, the attacker can call the `redeem()` function to use fxUSD tokens to redeem WBTC from the protocol. Since the f(x) system adopts the anchor price (lower than the actual market price) as the maximum price, the attacker can redeem more WBTC than they should. The price difference between the anchor price and the actual market price represents the attacker's potential profit.

Status

The development team modified the WBTC oracle pricing mechanism in commit 0fd9ff9.

When the WBTC-BTC price provided by Chainlink remains within the `maxWBTCDeviation` range, the protocol will use:

Anchor Price = max(WBTC-USD spot price, BTC-USD spot price)

If the Chainlink WBTC-BTC price deviates beyond the `maxWBTCDeviation` range, it is considered that WBTC has lost its peg. In this case, the protocol will use the WBTC-USD spot price as the Anchor Price. The principle behind this design is to always use the highest valid price for redeeming assets, ensuring the security of protocol assets.

4.3.5 Discussion on the function `_checkPositionDebtRatio()`.

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

Location

PositionOperateFlashLoanFacetV2.sol#L151

PositionOperateFlashLoanFacetV2.sol#L191

Description

When users call the `openOrAddPositionFlashLoanV2()` or `closeOrRemovePositionFlashLoanV2()` functions, they utilize flash loans to open, close, or adjust positions. These operations are executed internally through the `operate()` function, which already enforces upper and lower limits on position debt ratios, as seen in BasePool.sol#L170-L172. If the adjusted position does not meet the required debt ratio, the operation will fail.

However, the `openOrAddPositionFlashLoanV2()` function includes an additional `_checkPositionDebtRatio()` function, which redundantly revalidates the position's debt ratio. This raises the question of whether `_checkPositionDebtRatio()` serves a distinct purpose beyond the existing checks in `operate()`, or if it imposes an unnecessary restriction. Further clarification is needed regarding the function's role and necessity.

```
function onOpenOrAddPositionFlashLoanV2(
    address pool,
    uint256 position,
    uint256 amount,
    uint256 repayAmount,
    address recipient,
    bytes memory data
  ) external onlySelf {
 (bytes32 miscData, uint256 fxUSDAmount, address swapTarget, bytes
memory swapData) = abi.decode(
      data,
 (bytes32, uint256, address, bytes)
 );

    // 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));
    // @audit the `miscData` parameter is provided by the user
    _checkPositionDebtRatio(pool, position, miscData);
    IERC721(pool).transferFrom(address(this), recipient, position);

    emit OpenOrAdd(pool, position, recipient, amount, fxUSDAmount,
repayAmount);

    // swap fxUSD to collateral token
    _swap(fxUSD, IPool(pool).collateralToken(), fxUSDAmount,
repayAmount, swapTarget, swapData);
  }

function onCloseOrRemovePositionFlashLoanV2(
    address pool,
    uint256 position,
    uint256 amount,
    uint256 borrowAmount,
    address recipient,
    bytes memory data
  ) external onlySelf {
 (bytes32 miscData, uint256 fxUSDAmount, address swapTarget, bytes
memory swapData) = abi.decode(
      data,
 (bytes32, uint256, address, bytes)
 );

    // swap collateral token to fxUSD
    _swap(IPool(pool).collateralToken(), fxUSD, borrowAmount,
fxUSDAmount, swapTarget, swapData);

    // close or remove collateral from position
    IERC721(pool).transferFrom(recipient, address(this), position);
 (, uint256 maxFxUSD) = IPool(pool).getPosition(position);
    if (fxUSDAmount >= maxFxUSD) {
      // close entire position
      IPoolManager(poolManager).operate(pool, position,
type(int256).min, type(int256).min);
    } else {
      IPoolManager(poolManager).operate(pool, position, -int256(amount),
-int256(fxUSDAmount));
      // @audit the `miscData` parameter is provided by the user
      _checkPositionDebtRatio(pool, position, miscData);
    }
    IERC721(pool).transferFrom(address(this), recipient, position);

    emit CloseOrRemove(pool, position, recipient, amount, fxUSDAmount,
borrowAmount);
  }

/// @dev Internal function to check debt ratio for the position.
/// @param pool The address of fx position pool.
/// @param positionId The index of the position.
/// @param miscData The encoded data for debt ratio range.
```

```
function _checkPositionDebtRatio(address pool, uint256 positionId,
bytes32 miscData) internal view {
  uint256 debtRatio = IPool(pool).getPositionDebtRatio(positionId);
  uint256 minDebtRatio = miscData.decodeUint(0, 60);
  uint256 maxDebtRatio = miscData.decodeUint(60, 60);
  if (debtRatio < minDebtRatio || debtRatio > maxDebtRatio) {
    revert ErrorDebtRatioOutOfRange();
  }
```

Status

The development team explained that the `operate()` function only enforces the upper and lower limits of the debt ratio permitted by the protocol, whereas the `checkPositionDebtRatio()` function verifies the user's target leverage limits, which are generally stricter.

Since the `operate()` function involves token swaps and is subject to oracle price fluctuations, the actual leverage of a user's position may deviate from the intended leverage. The `checkPositionDebtRatio()` function helps prevent excessive deviations between actual and expected leverage. Additionally, its restrictions provide a certain level of protection against potential sandwich attack risks during the swap process.

5. Conclusion

After auditing and analyzing the upgraded version of the f(x) protocol, 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 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