Security Audit Report

Limit Order Module of f(x) Protocol

by AladdinDAO

SECBIT

October 15, 2025

1. Introduction

The AladdinDAO is a decentralized network to shift crypto investments from venture capitalists to the wisdom of crowds through collective value discovery. The f(x) Protocol is a decentralized system built to deliver high-leverage trading and yield-bearing stable coins without relying on centralized intermediaries. To address issues associated with large position operations—such as significant slippage, lack of price transparency, and uncertainty in asset conversion rates—the f(x) protocol introduces a limit order module. As an auxiliary component of the protocol, this module allows users to specify an expected conversion price, with the asset exchange executed at the predefined rate once market conditions meet the trigger price. SECBIT Labs conducted an audit from August 19, 2025, to October 15, 2025, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the Limit Order Module of f(x) Protocol 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 The `getTradeableOrder` function contains flaws in its logic for determining order tradability.	Info	Discussed
Design & Implementation	4.3.2 The `createAuctions` function design presents certain deficiencies.	Info	Discussed
Design & Implementation	4.3.3 The receiving address for the `buyToken` in auction orders is not configured, which prevents all such orders from being executed.	Medium	Fixed
Design & Implementation	4.3.4 Consideration should be given to adding a cancellation feature for auction orders to prevent unintended scenarios.	Info	Fixed
Design & Implementation	4.3.5 The `settleKillPool()` function can be invoked without first calling `killPool()`, which may result in burning long debt tokens without handling short pool bad debt.	Low	Fixed
Design & Implementation	4.3.6 The `settleKillPool()` function does not validate the consistency between `data.debts` and `data.settledDebts`, which may cause underflow and trigger a revert in cases involving asynchronous operations.	Info	Fixed
Design & Implementation	4.3.7 Discussion of `redeemForSettle()` function.	Medium	Fixed
Design & Implementation	4.3.8 The `DutchAuctionSwap` contract inherits from `AccessControlUpgradeable` but does not initialize it properly.	Info	Fixed
Design & Implementation	4.3.9 The purpose and intended use of the parameters `nonce` and `salt` should be clarified.	Info	Discussed
Design & Implementation	4.3.10 Discussion on the signature verification mechanism.	Info	Discussed
Design & Implementation	4.3.11 Rounding errors in certain calculations may cause associated order-matching operations to fail.	Medium	Fixed
Design & Implementation	4.3.12 When a limit order is partially filled, the corresponding position NFT must be re-approved for the `LimitOrderManager` contract.	Info	Discussed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the Limit Order Module of f(x) Protocol is shown below:

- Smart contract code
  - initial review commit *d5d3e46*
  - final review commit *edf00e9*

2.2 Contract List

The following content shows the contracts included in the Limit Order Module of f(x) Protocol, which the SECBIT team audits:

Name	Lines	Description
DutchAuctionSwap.sol	152	Provides asset auction services for the protocol administrator, who creates auction orders through this contract. Order matching is carried out via the Cow Protocol.
FxUSDBasePool.sol	460	Users can deposit fxUSD and USDC tokens into the protocol contract, where these assets are utilized for position leverage adjustment and protocol risk mitigation.
PoolConfiguration.sol	301	Fee configuration contracts that allow the protocol to set and manage various fee parameters.
PoolManager.sol	731	The peripheral contract provides direct interfaces for users to execute position operations, including long position opening, closing, adjustment, and liquidation procedures.
BasePool.sol	490	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	204	The supplementary module of the protocol calculates corresponding tick values based on position debt ratios, facilitating position liquidation processes through tick-based execution mechanisms.
ShortPoolManager.sol	515	The peripheral contract provides direct interfaces for users to execute short position operations, including position opening, closing, adjustment, and liquidation procedures.
BaseConditionalOrder.sol	31	Base logic for conditional orders.
GPv2Interaction.sol	41	Gnosis protocol v2 interaction library.
GPv2Order.sol	89	Gnosis protocol v2 order library.
LimitOrderManager.sol	220	The core contract of the limit order module. Makers publish limit order trade information off-chain, and takers execute the order on-chain.
OrderExecutionLibrary.sol	41	A supporting contract for the limit order module that provides basic order information.
OrderLibrary.sol	172	A supporting contract for the limit order module that calculates the expected asset amounts a maker will sell and receive.

Notice: This audit specifically focuses on the modified portions of the FxUSDBasePool.sol, PoolConfiguration.sol, PoolManager.sol, BasePool.sol, TickLogic.sol, and ShortPoolManager.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 Limit Order Module of the f(x) Protocol are the Governance Account and the Common Account.

- Governance Account
  - Description

    Contract Administrator, including default administrator role, emergency role, auction creator role, and auction collector role.
  - Authority
    - Transfer ownership
    - Create Cow protocol auction order
    - Withdraw auction funds
  - Method of Authorization

    The contract administrator is the contract's creator or authorized by transferring the governance account.
- Common Account
  - Description

    Users can either create a limit order as a maker or execute an existing order as a taker.
  - Authority
    - Submit a limit order off-chain
    - Execute a limit order on-chain
    - Cancel a partially unfilled limit order
  - Method of Authorization

    No authorization required

3.2 Functional Analysis

The f(x) Protocol v2.1 establishes a complete long and short mechanism. To mitigate slippage losses that may occur during position operating, the protocol introduces a limit order module. In addition, the limit order module also supports stop-loss and take-profit operations for positions. The SECBIT team conducted a detailed audit of some of the contracts in the protocol. We can divide the critical functions of the contract into two parts:

DutchAuctionSwap

The protocol administrator can use this contract to create auction orders, which are then broadcast and matched through the Cow Protocol. The main functions in these contracts are as follows:

- `createAuction()`

  The administrator creates an auction order through this function, which is then broadcast via the Cow Protocol.

- `createAuctions()`

  The administrator creates multiple auction orders in batches through this function, which are then broadcast via the Cow Protocol.

- `collectAuctions()`

  The administrator withdraws the funds from auction orders through this contract.

LimitOrderManager

This contract implements the core logic of the limit order functionality. A maker submits a limit order off-chain, which is then broadcast by the f(x) protocol. Any user may execute the order on-chain once the specified conditions are met. The main functions in this contract are as follows:

- `fillOrder()`

  The taker executes a maker's limit order through this function.

- `cancelOrder()`

  After a limit order is partially filled, the maker can cancel the remaining unfilled portion of the order through this function.

4. Audit Detail

This part describes the process, and the detailed audit results also demonstrate the problems and potential risks.

4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project for code bugs, logical implementation, and potential risks. The process consists of four steps:

- Full analysis of contract code line by line.
- Evaluation of vulnerabilities and potential risks revealed in the contract code.
- Communication on assessment and confirmation.
- Audit report writing.

4.2 Audit Result

After scanning with adelaide, sf-checker, and badmsg.sender (internal version) developed by SECBIT Labs and open source tools, including Mythril, Slither, SmartCheck, and Securify, the auditing team performed a manual assessment. The team inspected the contract line by line, and the result could be categorized into the following types:

Number	Classification	Result
1	Normal functioning of features defined by the contract	✓
2	No obvious bug (e.g., overflow, underflow)	✓
3	Pass Solidity compiler check with no potential error	✓
4	Pass common tools check with no obvious vulnerability	✓
5	No obvious gas-consuming operation	✓
6	Meet with ERC20 standard	✓
7	No risk in low-level call (call, delegatecall, callcode) and in-line assembly	✓
8	No deprecated or outdated usage	✓
9	Explicit implementation, visibility, variable type, and Solidity version number	✓
10	No redundant code	✓
11	No potential risk manipulated by timestamp and network environment	✓
12	Explicit business logic	✓
13	Implementation consistent with annotation and other info	✓
14	No hidden code about any logic that is not mentioned in the design	✓
15	No ambiguous logic	✓
16	No risk threatening the developing team	✓
17	No risk threatening exchanges, wallets, and DApps	✓
18	No risk threatening token holders	✓
19	No privilege on managing others' balances	✓
20	No non-essential minting method	✓
21	Correct managing hierarchy	✓

4.3 Issues

4.3.1 The `getTradeableOrder` function contains flaws in its logic for determining order tradability.

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

Location

DutchAuctionSwap.sol#L181-L183

Description

The `getTradeableOrder()` function is designed to query active orders created by the contract that have not yet been executed. However, consider the following scenario: a `sellToken` auction order created by the contract has already been executed through the Cow Protocol. The administrator has withdrawn the corresponding `buyToken` from the `receiver` address, while the order's validity period has not yet expired. In this case, when a user calls `getTradeableOrder()`, the order that has in fact been completed (and is effectively invalid) would still appear as active, which may lead to user misinterpretation.

In addition, if an external user directly transfers the full amount of `buyToken` to the `receiver` address, the order would be deemed non-tradeable. This behavior could constitute a denial-of-service (DoS) vector, with uncertain impact.

```
function getTradeableOrder(
    address,
    address,
    bytes32,
    bytes calldata staticInput,
    bytes calldata
  ) public view override returns (GPv2Order.Data memory order) {
    address receiver = auctionReceiver[keccak256(staticInput)];
    if (receiver == address(0)) revert InvalidAuction();

    Data memory data = abi.decode(staticInput, (Data));

    // woah there! you're too early and the auction hasn't started. Come back later.
    if (data.startTime > uint32(block.timestamp)) {
      revert PollTryAtEpoch(data.startTime, "auction not started");
    }

    // bucket is the current step of the auction, use unchecked here to save gas
    uint32 bucket;
    unchecked {
      bucket = uint32(block.timestamp - data.startTime) / data.stepDuration;
    }

    // if too late, not valid, revert
    // @audit it may be considered to limit the validity of the order
    if (bucket >= data.numSteps) {
      revert PollNever("auction ended");
    }

    // calculate the current buy amount
```

```
    // Note: due to integer rounding, the current buy amount might be slightly lower
than expected (off-by-one)
    uint256 bucketBuyAmount = data.startBuyAmount - (bucket * data.stepDiscount *
data.startBuyAmount) / 10000;

    // generate the order
    order = GPv2Order.Data(
      data.sellToken,
      data.buyToken,
      receiver,
      data.sellAmount,
      bucketBuyAmount,
      data.startTime + (bucket + 1) * data.stepDuration, // valid until the end of
the current bucket
      data.appData,
      0, // use zero fee for limit orders
      GPv2Order.KIND_SELL, // only sell order support for now
      false, // partially fillable orders are not supported
      GPv2Order.BALANCE_ERC20,
      GPv2Order.BALANCE_ERC20
    );

    // check if the auction is filled
    // @audit this condition will not apply if the administrator withdraws the funds
after the order is completed
    if (data.buyToken.balanceOf(receiver) >= bucketBuyAmount) {
      revert PollNever("auction filled");
    }
  }
```

Status

According to the development team, the `DutchAuctionSwap` contract is not intended to be exposed to external users. Its sole requirement is to enable internal token swaps via the CowSwap protocol. Therefore, broader user-facing scenarios and related complexities do not need to be considered.

4.3.2 The `createAuctions` function design presents certain deficiencies.

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

Location

DutchAuctionSwap.sol#L253

Description

The administrator can create multiple `sellToken` auction orders using the `createAuction()` or `createAuctions()` functions. When an order is created, this contract approves the Cow Protocol's `settlement` contract to spend the specified `sellAmount` of `sellToken`. It should be noted that when the `DutchAuctionSwap` contract creates multiple orders involving the same `sellToken`, the approval amount will be set to the value of the most recent order rather than the cumulative total of all orders. As a result, some orders may fail to execute due to insufficient allowance.

In addition, the `createAuction()` function shares a single `receiver` address across multiple auctions. The `buyToken` amounts received from different auctions are therefore aggregated, which prevents the `getTradeableOrder()` function from accurately determining the tradability of individual orders.

```
function createAuction(Data memory data, bytes32 salt) external
onlyRole(AUCTION_CREATOR_ROLE) {
    DutchAuctionReceiver receiver = new DutchAuctionReceiver(address(data.buyToken));

    _createAuction(data, address(receiver), salt);
  }

function createAuctions(Data[] memory data, bytes32[] memory salts) external
onlyRole(AUCTION_CREATOR_ROLE) {
    DutchAuctionReceiver receiver = new
DutchAuctionReceiver(address(data[0].buyToken));
    for (uint256 i = 0; i < data.length; i++) {
      if (i > 0 && data[i].buyToken != data[i - 1].buyToken) {
        revert InvalidAuction();
      }

      _createAuction(data[i], address(receiver), salts[i]);
    }
}


function _createAuction(Data memory data, address receiver, bytes32 salt) internal {
    _validateData(data);

    bytes memory staticInput = abi.encode(data);
    bytes32 staticInputHash = keccak256(staticInput);
    if (auctionReceiver[staticInputHash] != address(0)) revert
AuctionAlreadyExists();

    IConditionalOrder.ConditionalOrderParams memory params =
IConditionalOrder.ConditionalOrderParams({
      handler: this,
      salt: salt,
      staticInput: staticInput
    });
    IComposableCoW(composableCow).create(params, true);

    // approve sell token to cowswap
```

```
    // @audit note that when there are multiple identical sellToken orders, only the
authorization limit of the last order will be used and previous authorization
operations will be useless
    data.sellToken.forceApprove(settlement, data.sellAmount);

    emit AuctionCreated(staticInputHash, address(receiver));
}
```

```
// https://github.com/OpenZeppelin/openzeppelin-
contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol#L255
function forceApprove(IERC20 token, address spender, uint256 value) internal {
    if (!_safeApprove(token, spender, value, false)) {
        if (!_safeApprove(token, spender, 0, true)) revert
SafeERC20FailedOperation(address(token));
        if (!_safeApprove(token, spender, value, true)) revert
SafeERC20FailedOperation(address(token));
    }
}

// @audit each time the authorization function is executed, the quota is updated and
does not accumulate
function _safeApprove(IERC20 token, address spender, uint256 value, bool bubble)
private returns (bool success) {
    bytes4 selector = IERC20.approve.selector;

    assembly ("memory-safe") {
        let fmp := mload(0x40)
        mstore(0x00, selector)
        mstore(0x04, and(spender, shr(96, not(0))))
        mstore(0x24, value)
        success := call(gas(), token, 0, 0, 0x44, 0, 0x20)
        // if call success and return is true, all is good.
        // otherwise (not success or return is not true), we need to perform further
checks
        if iszero(and(success, eq(mload(0x00), 1))) {
            // if the call was a failure and bubble is enabled, bubble the error
            if and(iszero(success), bubble) {
                returndatacopy(fmp, 0, returndatasize())
                revert(fmp, returndatasize())
            }
            // if the return value is not true, then the call is only successful if:
            // - the token address has code
            // - the returndata is empty
            success := and(success, and(iszero(returndatasize()),
gt(extcodesize(token), 0)))
        }
        mstore(0x40, fmp)
    }
}
```

Status

According to the development team, this contract is not exposed to external users. When the protocol administrator uses this contract, they ensure that multiple active orders do not involve the same `sellToken`.

4.3.3 The receiving address for the `buyToken` in auction orders is not configured, which prevents all such orders from being executed.

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

Location

DutchAuctionSwap.sol#L238-L256

Description

When creating a `sellToken` auction order, the `DutchAuctionSwap` contract fails to set the `receiver` address corresponding to `auctionReceiver[staticInputHash]`. As a result, the `buyToken` for the order is sent to the zero address by default.

When the Cow Protocol processes the order, it calls the `getTradeableOrder()` function in the `DutchAuctionSwap` contract to determine if the order is valid. This function internally verifies that the `receiver` address is non-zero. Since the `receiver` is currently set to the zero address, the function will revert, preventing all such orders from being executed.

```
function _createAuction(Data memory data, address receiver, bytes32 salt) internal {
    _validateData(data);

    bytes memory staticInput = abi.encode(data);
    bytes32 staticInputHash = keccak256(staticInput);
    // @audit the auctionReceiver is not set at this time
    if (auctionReceiver[staticInputHash] != address(0)) revert
AuctionAlreadyExists();

    IConditionalOrder.ConditionalOrderParams memory params =
IConditionalOrder.ConditionalOrderParams({
      handler: this,
      salt: salt,
      staticInput: staticInput
    });
    IComposableCoW(composableCow).create(params, true);

    // approve sell token to cowswap
    data.sellToken.forceApprove(settlement, data.sellAmount);

    emit AuctionCreated(staticInputHash, address(receiver));
  }
```

Status

The development team has confirmed this issue and has fixed it in commit 9128e61.

4.3.4 Consideration should be given to adding a cancellation feature for auction orders to prevent unintended scenarios.

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

Description

The contract does not provide a corresponding order cancellation function such as `remove()` (see documentation). As a result, once an order is created, it cannot be actively canceled. Consider adding an order cancellation feature to mitigate potential unintended situations.

Status

The development team addressed this issue in commit 9128e61 by introducing the `cancelAuction()` function for order cancellation. In addition, the `getTradeableOrder()` function was updated to include a check on the current order status, thereby resolving the problem.

4.3.5 The `settleKillPool()` function can be invoked without first calling `killPool()`, which may result in burning long debt tokens without handling short pool bad debt.

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

Location

ShortPoolManager.sol#L459-L474

Description

In extreme market conditions, if the short pool becomes undercollateralized, the administrator may perform a global debt settlement. Specifically, the administrator first calls the `killPool()` function to disable the `liquidate()` function of the short pool, and then calls the `settleKillPool()` function (potentially multiple times) to liquidate the remaining collateral and handle the system bad debt of the short pool.

However, it should be noted that the administrator can currently call `settleKillPool()` directly without first invoking `killPool()`. This allows the function to burn debt tokens held in the long contract without addressing any short pool debt or bad debt, which is inconsistent with the intended settlement logic.

```
// @audit the administrator can also call the following function directly without
calling the killPool() function, which will directly cause all debt certificates held
by the long pool to be burned
function settleKillPool(address pool, uint256 colls) external
onlyRegisteredPool(pool) onlyRole(POOL_KILLER_ROLE) {
    KillPoolData memory data = killPoolData[pool];
```

```
    if (colls > data.colls - data.settledColls) colls = data.colls -
data.settledColls;

    address longPool = IShortPool(pool).counterparty();
    uint256 debts = ILongPoolManager(counterparty).redeemForSettle(longPool, colls);
    emit SettleKillPool(pool, colls, debts);

    data.settledColls += colls;
    data.settledDebts += debts;
    killPoolData[pool] = data;

    // @audit when this function is invoked directly, data.settledColls == data.colls
== 0, and the condition still holds
    if (data.settledColls == data.colls) {
      ILongPoolManager(counterparty).settleShortPool(longPool, pool, data.debts -
data.settledDebts);
    }
}
```

```
// https://github.com/AladdinDAO/fx-protocol-contracts-
internal/blob/d5d3e469f5f461be8d086e712fdd6e757a5cb0dc/contracts/core/PoolManager.sol
#L716-L736
function settleShortPool(
    address longPool,
    address shortPool,
    uint256 shortfall  // @audit bad debt (wsteth or wbtc)
  ) external onlyCounterparty onlyRegisteredPool(shortPool) nonReentrant {
    address collateralToken = ILongPool(longPool).collateralToken();
    uint256 scalingFactor = _getTokenScalingFactor(collateralToken);

    // @audit convert wsteth amount to steth amount
    // for wbtc, convert to decimals 18
    uint256 rawShortfall = _scaleUp(shortfall, scalingFactor);

    // reduce collateral in long pool
    // @audit bad debts are charged in the form of funding costs through long pools
    ILongPool(longPool).reduceCollateral(rawShortfall);
    // @audit funding costs are deducted
    _changePoolCollateral(longPool, -int256(shortfall), -int256(rawShortfall));

    // burn all credit note tokens in pool manager.
    // for tokens outside of pool manager, the short pool is killed, the tokens are
useless.
    // @audit as noted, the credit notes held by the user will not be able to use,
the debt instruments under the long contract and will be burned out
    address creditNote = IShortPool(shortPool).creditNote();
    uint256 creditNoteAmount = IERC20(creditNote).balanceOf(address(this));
    if (creditNoteAmount > 0) {
      ICreditNote(creditNote).burn(address(this), creditNoteAmount);
    }
}
```

Suggestion

It is recommended that the `settleKillPool()` function include a restriction that permits its execution only after `killPool()` has been called, to preserve the expected settlement process.

Status

The development team has confirmed this issue and has fixed it in commit 9128e61.

4.3.6 The `settleKillPool()` function does not validate the consistency between `data.debts` and `data.settledDebts`, which may cause underflow and trigger a revert in cases involving asynchronous operations.

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

Location

ShortPoolManager.sol#L472

Description

During global settlement of the short pool, the administrator must first call the `killPool()` function to halt liquidation operations, and then call the `settleKillPool()` function to distribute system bad debt. Specifically, when the administrator invokes `killPool()`, the function records the collateral amount held by the short pool (`killPoolData[pool].colls`) and the total debt (`killPoolData[pool].debts`). Subsequently, the administrator calls `settleKillPool()` to return the collateral from the short pool to the long pool, while the system bad debt of the short pool is offset through the collection of the long pool's funding cost.

Because `killPool()` and `settleKillPool()` are executed asynchronously, it cannot be guaranteed that the debt amount actually processed in `settleKillPool()` (`data.settledDebts`) will always be less than the total debt recorded in `data.debts`. As a result, the calculation `data.debts - data.settledDebts` may underflow and trigger a revert. It is recommended to add a validation check comparing the values of `data.debts` and `data.settledDebts` to ensure the settlement logic functions as intended.

```
function killPool(address pool) external onlyRegisteredPool(pool)
onlyRole(POOL_KILLER_ROLE) {
    // if the pool is already killed, this line will revert.
    (bool underCollateral, ) = IShortPool(pool).isUnderCollateral();
    if (!underCollateral) revert ErrorPoolNotUnderCollateral();

    uint256 rawDebts = IShortPool(pool).getTotalRawDebts();
    uint256 rawColls = IShortPool(pool).getTotalRawCollaterals();
    uint256 debts = _scaleDown(rawDebts,
_getTokenScalingFactor(IShortPool(pool).debtToken()));
    IShortPool(pool).kill();

    // @audit total collateral amount
```

```
        killPoolData[pool].colls = rawColls;
        // @audit total debt amount
        killPoolData[pool].debts = debts;

        emit KillPool(pool, rawColls, debts);
    }

    function settleKillPool(address pool, uint256 colls) external
    onlyRegisteredPool(pool) onlyRole(POOL_KILLER_ROLE) {
        KillPoolData memory data = killPoolData[pool];
        if (colls > data.colls - data.settledColls) colls = data.colls -
    data.settledColls;

        address longPool = IShortPool(pool).counterparty();
        uint256 debts = ILongPoolManager(counterparty).redeemForSettle(longPool, colls);
        emit SettleKillPool(pool, colls, debts);

        data.settledColls += colls;
        data.settledDebts += debts;
        killPoolData[pool] = data;

        if (data.settledColls == data.colls) {
        // @audit in extreme cases, it may be revert due to underflow
          ILongPoolManager(counterparty).settleShortPool(longPool, pool, data.debts -
    data.settledDebts);
        }
      }
```

Status

The development team has confirmed this issue and has fixed it in commit 9128e61.

4.3.7 Discussion of `redeemForSettle()` function.

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

Location

PoolManager.sol#L709-L710

Description

When the short pool becomes undercollateralized, the administrator must call the `settleKillPool()` function to convert the short pool's collateral (fxUSD tokens) into the long pool's collateral asset and return it to the corresponding long pool. This process is executed by the `redeemForSettle()` function in the `PoolManager` contract (long pool).

The internal logic of `redeemForSettle()` is as follows:

Based on the amount of fxUSD tokens provided by the short pool, calculate the corresponding amount of collateral (assumed to be wstETH) that can be redeemed from the long pool.

Deduct the corresponding amounts of wstETH and fxUSD from the long pool's internal ledger.

Since the fxUSD tokens provided by the short pool are used to repay the long pool's debt, the fxUSD tokens held by the short pool should be burned. The redemption of wstETH from the long pool by the short pool using fxUSD should be treated consistently with a normal user redemption through the `redeem()` function, meaning the fxUSD ledger should also be updated accordingly.

```
function settleKillPool(address pool, uint256 colls) external
onlyRegisteredPool(pool) onlyRole(POOL_KILLER_ROLE) {
    KillPoolData memory data = killPoolData[pool];
    if (colls > data.colls - data.settledColls) colls = data.colls -
data.settledColls;

    address longPool = IShortPool(pool).counterparty();

    // @audit this function redeems the collateral under the short position (fxusd)
from the long position and obtains the collateral under the long position. Since all
debt assets under the short position are provided by the long pool, these collateral
must be returned to the long pool.
    uint256 debts = ILongPoolManager(counterparty).redeemForSettle(longPool, colls);
    emit SettleKillPool(pool, colls, debts);

    data.settledColls += colls;
    data.settledDebts += debts;
    killPoolData[pool] = data;

    if (data.settledColls == data.colls) {
      ILongPoolManager(counterparty).settleShortPool(longPool, pool, data.debts -
data.settledDebts);
    }
}

function redeemForSettle(
    address pool,
    uint256 debts
  ) external onlyCounterparty onlyRegisteredPool(pool) nonReentrant returns (uint256
colls) {
    address collateralToken = ILongPool(pool).collateralToken();
    uint256 scalingFactor = _getTokenScalingFactor(collateralToken);

    // allow tick not moved to make sure all fxusd is redeemed
    (, uint256 rawColls) = ILongPool(pool).redeem(debts, true);
    colls = _scaleDown(rawColls, scalingFactor);

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

```
    // @audit in this process, the fxUSD provided by the short pool is already
converted into the collateral asset (wstETH) and returned to the long pool in the
form of corresponding debt tokens. therefore, the fxUSD should be directly burned,
and the fxUSD ledger should be updated accordingly at the same time
    _changePoolDebts(pool, -int256(debts));

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

Status

The development team has acknowledged the issue and resolved it in commit 2035599.

4.3.8 The `DutchAuctionSwap` contract inherits from `AccessControlUpgradeable` but does not initialize it properly.

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

Description

The contract does not assign an initial `DEFAULT_ADMIN_ROLE`. As a result, other roles cannot be configured appropriately, preventing the contract from functioning as intended.

Status

The development team confirmed that this contract is not intended to be used independently and marked it as an abstract contract in commit 9128e61.

4.3.9 The purpose and intended use of the parameters `nonce` and `salt` should be clarified.

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

Location

LimitOrderManager.sol#L151

Description

The `Order` struct contains three parameters intended to validate order validity: `nonce`, `salt`, and `deadline`. However, the code does not provide explicit documentation regarding their roles. Based on functionality, it appears that `nonce` and `deadline` may overlap. The intended purposes of these parameters need to be clarified, potentially as follows:

`nonce`: This parameter must be actively updated by the maker. Once updated, all previously submitted and pending orders from the maker become invalid, which facilitates batch order management.

`salt`: Used to uniquely identify an order when both `nonce` and `deadline` are the same. In theory, the combination of this parameter and `deadline` should be sufficient to guarantee the uniqueness and validity of an order.

`deadline`: The expiration timestamp of the order. The order remains valid until this time is reached.

```
// OrderLibrary.sol
struct Order {
    address maker; //@audit order initiator
    address pool;
    uint256 positionId;
    bool positionSide; // true for long, false for short
    bool orderSide; // true for open, false for close
    bool allowPartialFill; // true for partial fill, false for full fill
    uint256 triggerPrice; // if orderSide is true, order can be filled only oracle
price is <= triggerPrice; if orderSide is false, order can be filled only oracle
price is >= triggerPrice
    int256 fxUSDDelta; // positive: take from maker, negative: give to maker
    int256 collDelta; // collateral delta passing to the corresponding pool manager
    int256 debtDelta; // debt delta passing to the corresponding pool manager
    uint256 nonce;
    bytes32 salt;
    uint256 deadline;
    uint256 bonus; // bonus for taker in fxUSD
  }


function fillOrder(
    OrderLibrary.Order memory order,
    bytes memory signature,
    uint256 makingAmount,
    uint256 takingAmount
  ) external nonReentrant {

    // validate basic fields of the order.
    order.validateOrder();

    // check signature and nonce.
    bytes32 orderHash = _hash(order);
    _checkSignature(order, orderHash, signature);
    if (order.nonce != nonces(order.maker)) revert ErrOrderNonceExpired();

    // check order status
    OrderExecutionLibrary.Execution memory execution =
executions[orderHash].decode();
    if (execution.status != OrderExecutionLibrary.Status.New && execution.status !=
OrderExecutionLibrary.Status.PartialFilled) {
        revert ErrOrderCancelledOrFullyFilled();
    }

    // check making amount and taking amount.
    uint256 orderMakingAmount = order.getMakingAmount();
```

```
    uint256 orderTakingAmount = order.getTakingAmount();
    if (!order.allowPartialFill) {
      if (takingAmount < orderMakingAmount || makingAmount < orderTakingAmount) {
        revert ErrOrderCannotBeFullyFilled();
      }
    }
    // this is the actual taking amount and making amount from the caller
    uint256 actualTakingAmount = Math.min(takingAmount, orderMakingAmount -
execution.filled);
    uint256 actualMakingAmount = (actualTakingAmount * makingAmount) / takingAmount;
    uint256 minMakingAmount = (actualTakingAmount * orderTakingAmount) /
orderMakingAmount;
    if (actualMakingAmount < minMakingAmount) revert ErrInsufficientMakingAmount();

    // fill the order
    _fillOrder(order, execution, orderMakingAmount, minMakingAmount,
actualTakingAmount);
    emit FillOrder(orderHash, _msgSender(), actualTakingAmount, minMakingAmount);

    // update execution status
    execution.filled = uint128(execution.filled + actualTakingAmount);
    if (execution.filled == orderMakingAmount) {
      execution.status = OrderExecutionLibrary.Status.FullyFilled;

      // transfer created position to the maker when fill the order fully
      if (order.positionId == 0 && execution.positionId != 0) {
        IERC721(order.pool).transferFrom(address(this), order.maker,
execution.positionId);
      }
    } else {
      execution.status = OrderExecutionLibrary.Status.PartialFilled;
    }
    executions[orderHash] = OrderExecutionLibrary.encode(execution);
  }
```

Status

The development team has confirmed the functions of these parameters. The `nonce` parameter enables the maker to perform batch order cancellations. When utilizing this feature, the maker is required to invoke the corresponding `increaseNonce()` function proactively.

4.3.10 Discussion on the signature verification mechanism.

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

Location

LimitOrderManager.sol#L249-L260

Description

In the current implementation, a maker can create an off-chain limit order using either an EOA (Externally Owned Account) address or an SCW (Smart Contract Wallet, e.g., multisig wallets). The maker signs the order data, and when the taker executes the order on-chain, the `_checkSignature()` function verifies that the order has not been tampered with.

The relevant logic is as follows:

```
if ((signature.length != 65 && signature.length != 64) || ECDSA.recover(orderHash,
signature) != order.maker) { ... }
```

The above code addresses the following four scenarios:

When the order is signed directly by an EOA (Externally Owned Account) address, the expected signature length is 65 bytes. (The current OpenZeppelin library does not support the 64-byte compact signature format, which will be introduced in version v6.0. At present, using a 64-byte signature will cause the `ECDSA.recover()` function to revert. Therefore, the current limit order code treats any 64-byte signature as invalid.)

(1). If the signature length provided by the taker is 65 bytes, and the return value of `ECDSA.recover(orderHash, signature)` matches `order.maker`, it indicates that the order information has not been altered, and the verification passes.

(2). If the signature length provided by the taker is not 65 bytes (nor 64 bytes, which—as analyzed above—would revert), the code will further check whether a contract is deployed at the corresponding address:

```
if (order.maker.code.length > 0)
{ ...
}else {
        revert ErrBadSignature();
}
```

Since the maker in this case is an EOA address, `maker.code.length == 0`, and thus the verification fails.

When the order is initiated by an EOA but checked via an SCW (Smart Contract Wallet) address, the signature length is determined by the SCW contract itself, offering greater flexibility. Two cases can be considered:

(3). If the signature length is 65 bytes (or 64 bytes), the `ECDSA.recover()` function will attempt to interpret the signature. Since the underlying ecrecover function cannot distinguish whether the signature originates from an SCW address, it treats it as a standard ECDSA signature. The result will typically return the original EOA address rather than the SCW address.

(4). If the SCW-signed signature length is not 65 or 64 bytes, the system will proceed to verify the signature's validity using the `isValidSignature()` function defined in the corresponding SCW contract.

In summary, when an SCW address is used to check the order and the signature length is 65 or 64 bytes, the current `_checkSignature()` function logic will rely on the contract's `isValidSignature()` function to verify the signature.

With the introduction of the EIP-7702 proposal on Ethereum, EOA (Externally Owned Account) addresses can temporarily acquire smart contract wallet functionality, giving rise to new scenarios. In this situation, if an EOA address sets code for itself via ERC-7702, it can validate signatures in two ways:

- By directly providing a valid EOA signature for its own address, which can be verified through `ECDSA.recover()`.
- By providing a signature that conforms to the ERC-1271 `isValidSignature` interface implemented in its own contract code (e.g., where another EOA is designated to sign).

Both types of signatures are theoretically valid, and the choice rests with the EOA address itself. This dual-signature mechanism introduces flexibility but may also pose potential, as yet unidentified, risks. Currently, no specific threats to the protocol arising from this dual-signature model have been identified.

```
function fillOrder(
    OrderLibrary.Order memory order,
    bytes memory signature,
    uint256 makingAmount,
    uint256 takingAmount
  ) external nonReentrant {
    // validate basic fields of the order.
    order.validateOrder();

    // check signature and nonce.
    bytes32 orderHash = _hash(order);

    // @audit verify that order information signed by an EOA address or SCW address
has not been modified
    _checkSignature(order, orderHash, signature);
    if (order.nonce != nonces(order.maker)) revert ErrOrderNonceExpired();

    ......
  }


function _checkSignature(OrderLibrary.Order memory order, bytes32 orderHash, bytes
memory signature) internal view {

    // @audit for EOA addresses, validation can only be performed if the signature
data length is 65 bytes and the signature information has not been modified
(currently the openzeppelin library does not support short signatures of 64 bytes)
    // @audit for SCW addresses, such as multi-signature address, the signature
length should not be equal to 64 or 65
    if ((signature.length != 65 && signature.length != 64) ||
ECDSA.recover(orderHash, signature) != order.maker) {
      if (order.maker.code.length > 0) {
        bytes4 result = IERC1271(order.maker).isValidSignature(orderHash, signature);
        if (result != IERC1271.isValidSignature.selector) {
          revert ErrBadSignature();
        }
      } else {
        revert ErrBadSignature();
```

```
        }
      }
    }
```

Status

The development team has acknowledged the issue and confirmed that the dual-signature model does not pose any security risks to the protocol.

4.3.11 Rounding errors in certain calculations may cause associated order-matching operations to fail.

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

Location

LimitOrderManager.sol#L280-L281

LimitOrderManager.sol#L285

LimitOrderManager.sol#L309

Description

When a maker places an off-chain order, the taker may execute the trade through the `fillOrder()` function. During the operation of long and short open positions, transactions may fail due to rounding errors caused by precision limitations in calculating related funds. As both long and short contracts rely on the same function for opening and closing positions, the parameters involved carry different meanings depending on the specific operation. The issue is illustrated below with examples under the long open/short open scenarios.

(1). Case: long open operation Assume the collateral for the position is the wstETH token, and the debt issued by the protocol is the fxusd token. In this case, the maker submits a limit order to purchase wstETH tokens as collateral to open a position and, in doing so, incurs a debt obligation denominated in fxusd tokens. Because the protocol employs overcollateralization, the collateralized wstETH tokens alone cannot generate a sufficient amount of fxusd tokens to settle with the taker. Therefore, the maker must additionally provide `order.fxUSDDelta` in fxusd tokens.

Assume the following scenario: the maker submits an order to purchase 2 wstETH tokens to open a position, minting 6100 fxusd tokens, and is additionally required to provide 2000 fxusd tokens to the taker. The parameters are:

Collateral amount: $order.collDelta = 2 \times 10^{18}$ wstETH token ,

Debt amount: $order.debtDelta = 6100 \times 10^{18}$ fxusd token

Additional fxusd token required: $order.fxusdDelta = 2000 \times 10^{18}$ fxusd token

The taker partially fills the order by agreeing to purchase 3000 fxusd tokens. Thus, taker's actual received amount: $takingAmount = actualTakingAmount = 3000 \times 10^{18}$ fxusd token

Total fxusd offered by maker: $orderMakingAmount = 8100 \times 10^{18}$ fxusd token

Under the long open condition: $orderMakingAmount = order.debtDelta + order.fxusdDelta$

In theory, the relationship among `takingAmount`, `debtDelta`, and `fxUSDDelta` within `_fillOrder()` should be: $takingAmount = debtDelta + fxUSDDelta$

However, according to the implemented calculation logic:

$$\begin{aligned} debtDelta &= \frac{order.debtDelta \times takingAmount}{orderMakingAmount} \\ &= \frac{6100 \times 10^{18} \times 3000 \times 10^{18}}{8100 \times 10^{18}} \\ &= 2259\,259\,259\,259\,259\,259\,259 \end{aligned} \tag{1}$$

$$\begin{aligned} fxUSDDelta &= \frac{order.fxUSDDelta \times takingAmount}{orderMakingAmount} \\ &= \frac{2000 \times 10^{18} \times 3000 \times 10^{18}}{8100 \times 10^{18}} \\ &= 740\,740\,740\,740\,740\,740\,740 \end{aligned}$$

Thus:

$$debtDelta + fxUSDDelta = 2999\,999\,999\,999\,999\,999\,999 \tag{2}$$

In this scenario, the contract transfers a total of $\mathbf{2999.\,999\,999\,999\,999\,999\,999}$ **fxusd token**, while the taker expects to receive exactly 3000 fxusd tokens. As a result, the contract cannot reconcile the one-unit shortfall due to rounding, leading to transaction failure. Since this auxiliary contract should not retain any residual funds, the discrepancy directly impacts the success of the order execution.

```
// @audit Case: long open operation
function fillOrder(
    OrderLibrary.Order memory order,
    bytes memory signature,
    uint256 makingAmount,
    uint256 takingAmount
  ) external nonReentrant {
     ......

    // check making amount and taking amount.
    // @audit the total number of fxusd tokens the maker intends to sell
    uint256 orderMakingAmount = order.getMakingAmount();
    // @audit the number of wstETH tokens that the taker needs to pay
    uint256 orderTakingAmount = order.getTakingAmount();
    if (!order.allowPartialFill) {
      if (takingAmount < orderMakingAmount || makingAmount < orderTakingAmount) {
        revert ErrOrderCannotBeFullyFilled();
      }
    }
    // this is the actual taking amount and making amount from the caller
    // @audit 3000 fxusd token
    uint256 actualTakingAmount = Math.min(takingAmount, orderMakingAmount -
execution.filled);
    uint256 actualMakingAmount = (actualTakingAmount * makingAmount) / takingAmount;
```

```
    uint256 minMakingAmount = (actualTakingAmount * orderTakingAmount) /
orderMakingAmount;
    if (actualMakingAmount < minMakingAmount) revert ErrInsufficientMakingAmount();

    // fill the order
    _fillOrder(order, execution, orderMakingAmount, minMakingAmount,
actualTakingAmount);
    emit FillOrder(orderHash, _msgSender(), actualTakingAmount, minMakingAmount);

  ......
  }

function _fillOrder(
    OrderLibrary.Order memory order,
    OrderExecutionLibrary.Execution memory execution,
    uint256 orderMakingAmount, // @audit maker wants to sell 8,100 fxusd tokens
    uint256 makingAmount, // @audit 2 wstETH token
    uint256 takingAmount // @audit 3000 fxusd token
  ) internal {
    address makingToken = order.getMakingToken();
    address takingToken = order.getTakingToken();

    // calculate the delta of the order
    int256 collDelta = (order.collDelta * int256(takingAmount)) /
int256(orderMakingAmount);
    // @audit calculate the number of fxusd tokens to mint for opening position
    int256 debtDelta = (order.debtDelta * int256(takingAmount)) /
int256(orderMakingAmount);
    // @audit calculate the additional number of fxusd tokens that maker needs to pay
    int256 fxUSDDelta = (order.fxUSDDelta * int256(takingAmount)) /
int256(orderMakingAmount);
    uint256 bonus = (order.bonus * takingAmount) / orderMakingAmount;

    // transfer taking token from caller to this contract
    IERC20(takingToken).safeTransferFrom(_msgSender(), address(this), makingAmount);
    // transfer taking token from order.maker to this contract
    // @audit maker will transfer fxusd token under this contract
    if (fxUSDDelta > 0) {
      IERC20(fxUSD).safeTransferFrom(order.maker, address(this),
uint256(fxUSDDelta));
    }
    // transfer position from order.maker to this contract
    if (order.positionId != 0) {
      execution.positionId = uint32(order.positionId);
      IERC721(order.pool).transferFrom(order.maker, address(this), order.positionId);
    }

    // operate the position
    address manager = order.positionSide ? LongPoolManager : ShortPoolManager;
    if (collDelta > 0) {
      IERC20(takingToken).forceApprove(manager, uint256(collDelta));
    }
```

```
    if (debtDelta < 0) {
      IERC20(takingToken).forceApprove(manager, uint256(-debtDelta));
    }

    // @audit when opening a position, it will mint the amount of debtDelta in fxusd
to the address
    execution.positionId = uint32(
      IPoolManager(manager).operate(order.pool, execution.positionId, collDelta,
debtDelta)
    );

    // transfer making token to the caller
    // @audit transfer fxusd token to the taker
    IERC20(makingToken).safeTransfer(_msgSender(), takingAmount);
    // transfer bonus from order.maker to the caller
    if (bonus > 0) {
      IERC20(fxUSD).safeTransferFrom(order.maker, _msgSender(), bonus);
    }
    // transfer position to the order.maker
    if (order.positionId != 0) {
      IERC721(order.pool).transferFrom(address(this), order.maker, order.positionId);
    }
    // transfer fxUSD to the order.maker
    if (fxUSDDelta < 0) {
      IERC20(fxUSD).safeTransfer(order.maker, uint256(-fxUSDDelta));
    }
  }
```

(2). Case: short open operation

Assume the collateral for the position is the fxusd token, and the debt is the wstETH token. The maker submits a limit order to purchase the required collateral (fxusd tokens) to open a short position, and in return receives debt in wstETH tokens. Because the protocol applies overcollateralization, the maker must provide additional fxusd tokens (`order.fxUSDDelta`) to complete the position opening.

Assume the following scenario: the maker places an order to sell 2.1 wstETH tokens (debt) obtained from opening a position. The taker is required to provide 8000 fxusd tokens to purchase these wstETH tokens, while the maker must additionally contribute 4600 fxusd tokens for the position opening. The parameters are:

Collateral amount: $order.collDelta = 12{,}600 \times 10^{18}$ fxusd token

Debt amount: $order.debtDelta = 2.1 \times 10^{18}$ wstETH token

Additional fxusd token required: $order.fxusdDelta = 4600 \times 10^{18}$ fxusd token

Total fxusd required from taker: $orderTakingAmount = 8000 \times 10^{18}$ fxusd token

The taker partially fills the order by purchasing 1.1 wstETH tokens. Thus:

Taker's received amount: $takingAmount = actualTakingAmount = 1.1 \times 10^{18}$ wstETH token

Maker's total offered debt amount: $MakingAmount = 2.1 \times 10^{18}$ wstETH token

Under the short open condition: $orderMakingAmount = order.debtDelta$

The amount of fxusd token that the taker needs to provide to purchase 1.1×10^{18} **wstETH token** is:

$$\begin{aligned} minMakingAmount &= makingAmount \\ &= \frac{actualTakingAmount \times orderTakingAmount}{orderMakingAmount} \\ &= \frac{1.1 \times 10^{18} \times 8000 \times 10^{18}}{2.1 \times 10^{18}} \\ &= 4\,190\,476\,190\,476\,190\,476\,190 \end{aligned} \tag{3}$$

The maker must additionally provide a proportional amount of fxusd token:

$$\begin{aligned} fxusdDelta &= \frac{order.fxUSDDelta \times takingAmount}{orderMakingAmount} \\ &= \frac{4600 \times 10^{18} \times 1.1 \times 10^{18}}{2.1 \times 10^{18}} \\ &= 2\,409\,523\,809\,523\,809\,523\,809 \end{aligned} \tag{4}$$

The actual collateral required for this partial fill is:

$$\begin{aligned} collDelta &= \frac{order.collDelta \times takingAmount}{orderMakingAmount} \\ &= \frac{12600 \times 10^{18} \times 1.1 \times 10^{18}}{2.1 \times 10^{18}} \\ &= 6600 \times 10^{18} \end{aligned} \tag{5}$$

In theory, the relationship among `collDelta`, `makingAmount`, and `fxUSDDelta` within `_fillOrder()` should be:

$$collDelta = makingAmount + fxUSDDelta \tag{6}$$

In practice, due to rounding errors, the calculation yields:

$$\begin{aligned} &makingAmount + fxUSDDelta \\ &= 4\,190\,476\,190\,476\,190\,476\,190 + 2\,409\,523\,809\,523\,809\,523\,809 \\ &= 6\,599\,999\,999\,999\,999\,999\,999 \\ &< collDelta \end{aligned} \tag{7}$$

As a result, the combined fxusd tokens provided by the taker and the maker are one unit short of the required collateral amount, causing the short open transaction to fail due to insufficient funds.

```
// @audit Case: short open operation
function fillOrder(
    OrderLibrary.Order memory order,
    bytes memory signature,
    uint256 makingAmount,
    uint256 takingAmount
  ) external nonReentrant {
     ......

    // check making amount and taking amount.
```

```
    // @audit the total amount of wstETH tokens the maker intends to sell
    uint256 orderMakingAmount = order.getMakingAmount();
    // @audit the number of fxusd tokens that the taker needs to pay
    uint256 orderTakingAmount = order.getTakingAmount();
    if (!order.allowPartialFill) {
      if (takingAmount < orderMakingAmount || makingAmount < orderTakingAmount) {
        revert ErrOrderCannotBeFullyFilled();
      }
    }
    // this is the actual taking amount and making amount from the caller
    // @audit  1.1 wsteth token
    uint256 actualTakingAmount = Math.min(takingAmount, orderMakingAmount -
execution.filled);
    uint256 actualMakingAmount = (actualTakingAmount * makingAmount) / takingAmount;
    uint256 minMakingAmount = (actualTakingAmount * orderTakingAmount) /
orderMakingAmount;
    if (actualMakingAmount < minMakingAmount) revert ErrInsufficientMakingAmount();

    // fill the order
    _fillOrder(order, execution, orderMakingAmount, minMakingAmount,
actualTakingAmount);
    emit FillOrder(orderHash, _msgSender(), actualTakingAmount, minMakingAmount);

  ......
  }

function _fillOrder(
    OrderLibrary.Order memory order,
    OrderExecutionLibrary.Execution memory execution,
    uint256 orderMakingAmount, // @audit maker wants to sell 2.1 wstETH tokens
    uint256 makingAmount, // @audit the fxusd token that the taker needs to pay
    uint256 takingAmount // @audit taker gets 1.1 wsteth tokens
  ) internal {
    address makingToken = order.getMakingToken();
    address takingToken = order.getTakingToken();

    // calculate the delta of the order
    // @audit the actual amount of fxusd tokens collateral required to be paid by the
taker for this order
    int256 collDelta = (order.collDelta * int256(takingAmount)) /
int256(orderMakingAmount);

    int256 debtDelta = (order.debtDelta * int256(takingAmount)) /
int256(orderMakingAmount);
    // @audit calculate the additional number of fxusd tokens that the maker needs to
pay
    int256 fxUSDDelta = (order.fxUSDDelta * int256(takingAmount)) /
int256(orderMakingAmount);
    uint256 bonus = (order.bonus * takingAmount) / orderMakingAmount;

    // transfer taking token from caller to this contract
    // @audit the number of fxusd tokens that the taker needs to pay
```

```
    IERC20(takingToken).safeTransferFrom(_msgSender(), address(this), makingAmount);
    // transfer taking token from order.maker to this contract
    // @audit the maker transfers fxusd token under this contract
    if (fxUSDDelta > 0) {
      IERC20(fxUSD).safeTransferFrom(order.maker, address(this),
uint256(fxUSDDelta));
    }
    // transfer position from order.maker to this contract
    if (order.positionId != 0) {
      execution.positionId = uint32(order.positionId);
      IERC721(order.pool).transferFrom(order.maker, address(this), order.positionId);
    }

    // operate the position
    address manager = order.positionSide ? LongPoolManager : ShortPoolManager;
    if (collDelta > 0) {
      IERC20(takingToken).forceApprove(manager, uint256(collDelta));
    }
    if (debtDelta < 0) {
      IERC20(takingToken).forceApprove(manager, uint256(-debtDelta));
    }

    // @audit it need to withdraw the 'collDelta' number of fxusd tokens from this
address
    execution.positionId = uint32(
      IPoolManager(manager).operate(order.pool, execution.positionId, collDelta,
debtDelta)
    );

    // transfer making token to the caller
    // @audit transfer the wsteth token to the taker
    IERC20(makingToken).safeTransfer(_msgSender(), takingAmount);
    // transfer bonus from order.maker to the caller
    if (bonus > 0) {
      IERC20(fxUSD).safeTransferFrom(order.maker, _msgSender(), bonus);
    }
    // transfer position to the order.maker
    if (order.positionId != 0) {
      IERC721(order.pool).transferFrom(address(this), order.maker, order.positionId);
    }
    // transfer fxUSD to the order.maker
    if (fxUSDDelta < 0) {
      IERC20(fxUSD).safeTransfer(order.maker, uint256(-fxUSDDelta));
    }
  }
```

Status

The development team has acknowledged the issue and resolved it in commit edf00e9c.

4.3.12 When a limit order is partially filled, the corresponding position NFT must be re-approved for the `LimitOrderManager` contract.

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

Location

LimitOrderManager.sol#L315-L317

Description

For limit order operations involving an existing position NFT (including opening, adjusting, or closing positions), the maker must first grant approval of the position NFT to the `LimitOrderManager` contract. When the taker calls the `fillOrder()` function to execute the order, the function transfers the maker's position NFT to the `LimitOrderManager` contract, invokes the relevant manager's `operate()` function to process the position, and then transfers the position NFT back to the maker's address.

Suppose the maker permits partial fills. In that case, each execution of a partial fill through the `fillOrder()` function requires the maker to re-authorize the position NFT to the `LimitOrderManager` contract before the remaining funds in the limit order can be processed. As a result, a taker cannot continuously execute multiple partial fills against the same limit order without the maker re-approving the position NFT after each fill. Confirmation is required as to whether this behavior is consistent with the intended design.

```
function _fillOrder(
    OrderLibrary.Order memory order,
    OrderExecutionLibrary.Execution memory execution,
    uint256 orderMakingAmount,
    uint256 makingAmount,
    uint256 takingAmount
  ) internal {

    ......

    // transfer making token to the caller
    IERC20(makingToken).safeTransfer(_msgSender(), takingAmount);
    // transfer bonus from order.maker to the caller
    if (bonus > 0) {
      IERC20(fxUSD).safeTransferFrom(order.maker, _msgSender(), bonus);
    }
    // transfer position to the order.maker
    // @audit for old positions, the position NFT under the contract will be
transferred to the maker
    if (order.positionId != 0) {
      IERC721(order.pool).transferFrom(address(this), order.maker, order.positionId);
    }
    // transfer fxUSD to the order.maker
    if (fxUSDDelta < 0) {
      IERC20(fxUSD).safeTransfer(order.maker, uint256(-fxUSDDelta));
    }
}
```

Status

The development team has confirmed the use of the `setApprovalForAll()` function. This function grants the limit order contract full operational authority over all NFT positions held by the maker, thereby addressing the issue of requiring repeated authorizations from the maker in cases of partial fulfillment of limit orders.

5. Conclusion

After auditing and analyzing the Limit Order Module of 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 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