Security Audit Report

f(x) Protocol by AladdinDAO

SECBIT

June 14, 2023

1. Introduction

The AladdinDAO is a decentralized network to shift crypto investments from venture capitalists to the wisdom of crowds through collective value discovery. As a part of the AladdinDAO ecosystem, the f(x) protocol creates two new ETH derivative assets, one with stablecoin-like low volatility called fractional ETH (fETH) and the second a leveraged long ETH perpetual token called leveraged ETH (xETH). SECBIT Labs conducted an audit from May 12 to June 14, 2023, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the f(x) contract has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising (see part 4 for details).

Type	Description	Level	Status
Gas optimization	4.3.1 Adding parameter checks during fund redemption to save gas.	Info	Fixed
Design & Implementation	4.3.2 Users cannot mint fToken and xToken simultaneously after the system officially launches.	Info	Discussed
Design & Implementation	4.3.3 A malicious user calling the `mintFToken()` function before calling the `mint()` function can disrupt the normal operation of the system.	Info	Discussed
Design & Implementation	4.3.4 Potential flash loan attack to siphon system rewards.	Low	Discussed

Design & Implementation	4.3.5 Potential flash loan attacks to seize the system's liquidation rewards.	Low	Discussed
Design & Implementation	4.3.6 The current code does not take into account the impact of platform fees on the actual principal amount deposited into the contract.	Info	Discussed
Design & Implementation	4.3.7 From a user's perspective, the functions `mintXToken()` and `addBaseToken()` may have overlapping functionalities under specific conditions.	Info	Discussed
Design & Implementation	4.3.8 Risk Discussion of the `redeem()` Function.	Low	Discussed
Design & Implementation	4.3.9 Parameter `_amount` assignment error.	Medium	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

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

- Smart contract code
    - initial review commit *2e07409*
    - final review commit *7636830*

2.2 Contract List

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

Name	Lines	Description
ETHGateway.sol	94	An auxiliary contract is designed to assist users in interacting with the f(x) protocol in a more user-friendly manner.
FractionalToken.sol	51	The Fractional ETH contract is derived from the basic ERC20 contract with modifications.
LeveragedToken.sol	43	The Leveraged ETH contract is derived from the basic ERC20 contract with modifications.
Market.sol	469	A contract that implements the core logic of the f(x) protocol.
StableCoinMath.sol	153	A contract that implements the core mathematical calculations of the f(x) protocol.
Treasury.sol	343	A contract to store the baseToken, where the core functions can only be called by the Market contract.

3. Contract Analysis

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

3.1 Role Classification

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

- Governance Account
    - Description

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

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

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

      No authorization required

3.2 Functional Analysis

The f(x) protocol implements a decentralized quasi-stablecoin with high collateral utilization efficiency, coupled with leveraged contracts that have low liquidation risks and no funding costs. 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:

ETHGateway

As a peripheral auxiliary contract, this contract provides users with a more user-friendly interface. The main functions in this contract are as below:

- `mintFToken()`

  Anyone could call this function to mint some fTokens with some Ethers under certain conditions.

- `mintXToken()`

  Anyone could call this function to mint some xTokens with some Ethers under certain conditions.

- `addBaseToken()`

  This function allows users to add baseTokens, minting xTokens in the process. At the same time, it adjusts the system's Net Asset Value (NAV) to increase the collateralization ratio.

- `redeem()`

  Users can redeem fTokens and xTokens and receive baseTokens through this function.

- `liquidate()`

  When the collateralization ratio is too low, users can use this function to perform liquidation, which helps to increase the system's collateralization ratio. Additionally, users will receive a certain percentage of rewards as incentives.

Market, StableCoinMath, and Treasury

These contracts implement the core functionality of the f(x) protocol. The main functions in these contracts are as below:

- `mint()`

  This function can only be called once during the initialization of the protocol to ensure the normal operation of the contract.

- `mintFToken()`

This function is used to mint fTokens, which may result in a decrease in the system's collateralization ratio.

- `mintXToken()`

  This function is used to mint xTokens, which increases the system's collateralization ratio and contributes to the stability of the system.

- `addBaseToken()`

  Users are allowed to call this function to add more baseTokens when the collateralization ratio falls below the specific threshold, aiming to maintain system stability. Additionally, users can earn rewards for their participation.

- `redeem()`

  By using this function, users can burn fTokens and xTokens, and in return, they will receive the corresponding baseTokens.

- `liquidate()`

  When the system's collateralization ratio falls below the liquidation threshold, users can utilize this function to perform asset liquidation, and they will also receive a certain percentage of rewards as incentives.

- `selfLiquidate()`

  When the system's collateralization ratio further decreases, the protocol administrator can call this function to perform asset liquidation, preventing the protocol from entering recapping mode.

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 Adding parameter checks during fund redemption to save gas.

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

Description

The function `redeem()` is used to redeem users' principal. By analyzing `Market.redeem()`, this function only allows the redemption of one type of token, meaning that either `_fTokenIn` or `_xTokenIn` must have a value of 0. However, the current `ETHGateway.redeem()` function does not restrict the values of input parameters `_fTokenIn` and `_xTokenIn`, which means users could potentially input both fToken and xToken, resulting in a failed redemption. Additionally, when one of the parameters `_fTokenIn` or `_xTokenIn` has a value of 0, executing the corresponding `_transferTokenIn()` and `_refund()` functions would also waste gas.

```
// @audit located in ETHGateway.sol
function redeem(
    uint256 _fTokenIn,
    uint256 _xTokenIn,
    uint256 _minBaseOut
  ) external returns (uint256 _baseOut) {
    _fTokenIn = _transferTokenIn(fToken, _fTokenIn);
    _xTokenIn = _transferTokenIn(xToken, _xTokenIn);

    _baseOut = IMarket(market).redeem(_fTokenIn, _xTokenIn,
address(this), _minBaseOut);

    _refund(fToken, msg.sender);
    _refund(xToken, msg.sender);
    _transferETH(_baseOut, msg.sender);
  }
```

```
// @audit located in Market.sol
function redeem(
    uint256 _fTokenIn,
    uint256 _xTokenIn,
    address _recipient,
    uint256 _minBaseOut
  ) external override nonReentrant cachePrice returns (uint256
_baseOut) {
    require(!redeemPaused, "redeem is paused");

    if (_fTokenIn == uint256(-1)) {
      _fTokenIn =
IERC20Upgradeable(fToken).balanceOf(msg.sender);
    }
    if (_xTokenIn == uint256(-1)) {
      _xTokenIn =
IERC20Upgradeable(xToken).balanceOf(msg.sender);
    }

    // @audit the following conditions require that either
fTokenIn or xTokenIn must have a value greater than 0, and the
other must be equal to 0.
    require(_fTokenIn > 0 || _xTokenIn > 0, "redeem zero
amount");
    require(_fTokenIn == 0 || _xTokenIn == 0, "only redeem
single side");

    ITreasury _treasury = ITreasury(treasury);
    MarketConfig memory _marketConfig = marketConfig;

   ......
  }
```

Suggestion

Restrict the parameters `_fTokenIn` and `_xTokenIn` with the following code:

```
function redeem(
        uint256 _fTokenIn,
        uint256 _xTokenIn,
        uint256 _minBaseOut
    ) external returns (uint256 _baseOut) {

        // @audit adjust the following code
        if(_fTokenIn > 0){
            _fTokenIn = _transferTokenIn(fToken, _fTokenIn);
            _xTokenIn = 0;
        }else{
            _fTokenIn = 0;
            _xTokenIn = _transferTokenIn(xToken, _xTokenIn);
        }

        _baseOut = IMarket(market).redeem(_fTokenIn, _xTokenIn,
address(this), _minBaseOut);

        // @audit adjust the following code
        if(_fTokenIn > 0){
            _refund(fToken, msg.sender);
        }else{
            _refund(xToken, msg.sender);
        }

        _transferETH(_baseOut, msg.sender);
    }
```

Status

The team has adopted the suggestion and fixed this issue in commit *7636830*.

4.3.2 Users cannot mint fToken and xToken simultaneously after the system officially launches.

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

Description

Please note that according to the protocol, the minting of both fTokens and xTokens can only be done using the `mint()` function. It is also observed that the `mint()` function requires `_treasury.totalBaseToken() == 0`, which means that the logic with `_baseSupply > 0` in the `mint()` function of the `Treasury.sol` contract will never be executed. This implies that users are unable to mint fTokens and xTokens simultaneously after the protocol initialization. It is necessary to confirm if this logic aligns with the intended design.

```
//@audit located in Market.sol
function mint(
    uint256 _baseIn,
    address _recipient,
    uint256 _minFTokenMinted,
    uint256 _minXTokenMinted
  ) external override nonReentrant cachePrice returns (uint256
_fTokenMinted, uint256 _xTokenMinted) {
    address _baseToken = baseToken;
    if (_baseIn == uint256(-1)) {
      _baseIn =
IERC20Upgradeable(_baseToken).balanceOf(msg.sender);
    }
    require(_baseIn > 0, "mint zero amount");

    ITreasury _treasury = ITreasury(treasury);
    require(_treasury.totalBaseToken() == 0, "only initialize
once");

    IERC20Upgradeable(_baseToken).safeTransferFrom(msg.sender,
address(_treasury), _baseIn);
    (_fTokenMinted, _xTokenMinted) = _treasury.mint(_baseIn,
_recipient, ITreasury.MintOption.Both);

    ......
  }
```

```
// @audit located in Treasury.sol
function mint(
    uint256 _baseIn,
    address _recipient,
    MintOption _option
  ) external override onlyMarket returns (uint256 _fTokenOut,
uint256 _xTokenOut) {
    StableCoinMath.SwapState memory _state = _loadSwapState();

    if (_option == MintOption.FToken) {
      _fTokenOut = _state.mintFToken(_baseIn);
    } else if (_option == MintOption.XToken) {
      _xTokenOut = _state.mintXToken(_baseIn);
    } else {
      if (_state.baseSupply == 0) {
        ......
      } else {
        (_fTokenOut, _xTokenOut) = _state.mint(_baseIn);
      }
    }

    .......
  }

//@audit located in StableCoinMath.sol
function mint(SwapState memory state, uint256 _baseIn)
  internal
  pure
  returns (uint256 _fTokenOut, uint256 _xTokenOut)
{
  //  n * v = nf * vf + nx * vx
  //  (n + dn) * v = (nf + df) * vf + (nx + dx) * vx
  //  ((nf + df) * vf) / ((n + dn) * v) = (nf * vf) / (n * v)
  //  ((nx + dx) * vx) / ((n + dn) * v) = (nx * vx) / (n * v)
  // =>
  //   df = nf * dn / n
  //   dx = nx * dn / n
  _fTokenOut =
state.fSupply.mul(_baseIn).div(state.baseSupply);
```

```
    _xTokenOut =
  state.xSupply.mul(_baseIn).div(state.baseSupply);
   }
```

Status

The team has confirmed that the current design does not allow the simultaneous minting of fToken and xToken.

4.3.3 A malicious user calling the `mintFToken()` function before calling the `mint()` function can disrupt the normal operation of the system.

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

Description

According to the protocol design logic, the `mint()` function should be called by any user to initialize the protocol. However, we have noticed that after the contract is deployed, any user can preemptively call the `mintFToken()` function, which would directly prevent the protocol from being used. Consider the following scenario: a malicious user calls the `mintFToken()` function before calling the `mint()` function and transfers a very small amount of baseToken to the contract. Assuming that the `fTokenMintInSystemStabilityModePaused` parameter is false during initialization (in practice, this value is only set to true by the DAO in emergencies), the collateralization ratio check will be bypassed. Since there are baseTokens in the contract, `Treasury.totalUnderlying() > 0`, and the `mint()` function cannot be executed, preventing the protocol from being initialized. This risk only occurs during protocol initialization and can be mitigated by ensuring that the `mint()` function is called first by a user.

```
  // @audit located in Market.sol
```

```
function mint(
    uint256 _baseIn,
    address _recipient,
    uint256 _minFTokenMinted,
    uint256 _minXTokenMinted
  ) external override nonReentrant cachePrice returns (uint256
_fTokenMinted, uint256 _xTokenMinted) {
    address _baseToken = baseToken;
    if (_baseIn == uint256(-1)) {
      _baseIn =
IERC20Upgradeable(_baseToken).balanceOf(msg.sender);
    }
    require(_baseIn > 0, "mint zero amount");

    ITreasury _treasury = ITreasury(treasury);
    //@audit if the mintFToken() function is called
preemptively, the following condition will not be met
    require(_treasury.totalBaseToken() == 0, "only initialize
once");

    IERC20Upgradeable(_baseToken).safeTransferFrom(msg.sender,
address(_treasury), _baseIn);
    (_fTokenMinted, _xTokenMinted) = _treasury.mint(_baseIn,
_recipient, ITreasury.MintOption.Both);

    require(_fTokenMinted >= _minFTokenMinted, "insufficient
fToken output");
    require(_xTokenMinted >= _minXTokenMinted, "insufficient
xToken output");

    emit Mint(msg.sender, _recipient, _baseIn, _fTokenMinted,
_xTokenMinted, 0);
  }

  /// @inheritdoc IMarket
  function mintFToken(
    uint256 _baseIn,
    address _recipient,
    uint256 _minFTokenMinted
  ) external override nonReentrant cachePrice returns (uint256
_fTokenMinted) {
    require(!mintPaused, "mint is paused");

    address _baseToken = baseToken;
    if (_baseIn == uint256(-1)) {
      _baseIn =
IERC20Upgradeable(_baseToken).balanceOf(msg.sender);
    }
    require(_baseIn > 0, "mint zero amount");

    ITreasury _treasury = ITreasury(treasury);

    // @audit In this case, the return value here would be 0.
    (uint256 _maxBaseInBeforeSystemStabilityMode, ) =
_treasury.maxMintableFToken(marketConfig.stabilityRatio);

    if (fTokenMintInSystemStabilityModePaused) {
      uint256 _collateralRatio = _treasury.collateralRatio();
      require(_collateralRatio > marketConfig.stabilityRatio,
"fToken mint paused");

      // bound maximum amount of base token to mint fToken.
      if (_baseIn > _maxBaseInBeforeSystemStabilityMode) {
        _baseIn = _maxBaseInBeforeSystemStabilityMode;
      }
    }

    uint256 _amountWithoutFee = _deductFTokenMintFee(_baseIn,
fTokenMintFeeRatio, _maxBaseInBeforeSystemStabilityMode);

    IERC20Upgradeable(_baseToken).safeTransferFrom(msg.sender,
address(_treasury), _amountWithoutFee);
    (_fTokenMinted, ) = _treasury.mint(_amountWithoutFee,
_recipient, ITreasury.MintOption.FToken);
    require(_fTokenMinted >= _minFTokenMinted, "insufficient
fToken output");
```

```
    emit Mint(msg.sender, _recipient, _baseIn, _fTokenMinted,
0, _baseIn - _amountWithoutFee);
  }
```

Status

The team confirms that they will immediately call the `mint()` function after deploying the protocol. On the other hand, if an unfortunate attack occurs, the team will redeploy the contracts.

4.3.4 Potential flash loan attack to siphon system rewards.

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

Description

When the protocol is operating normally, users can utilize the `mintFToken()` function to deposit baseTokens and mint fTokens, which lowers the system's collateralization ratio. If the collateralization ratio falls below the warning line, any user can call the `addBaseToken()` function to add more baseTokens to the system and unilaterally mint new xTokens. In this scenario, the value of the xTokens they receive will be greater than the value of the baseTokens they input.

Current code logic may have a potential risk of flash loan attacks. Consider the following scenario:

(1). In a normal functioning system, an attacker can use the `mintFToken()` function to mint fToken. Assuming that the parameter `fTokenMintInSystemStabilityModePaused` is false during initialization (in practice, this value is typically set to true only in emergencies by the DAO), the collateral ratio check will be skipped. In this case, the attacker can manipulate the collateralization ratio by minting fToken. The lower the collateral ratio (but not below the liquidation ratio), the more obvious the attack effect.

(2). The attacker can call the `addBaseToken()` function to add baseToken to the system. It is important to note that the `addBaseToken()` function calculates the number of funds a user should deposit based on the current stable collateral ratio to prevent improper profits. Therefore, the attacker can exploit flash loans to borrow baseToken and then call the `addBaseToken()` function to add collateral to the system, raising the collateral ratio to the stable line. In the system's design logic, adding collateral to the system through the `addBaseToken()` function provides incentives proportionally. If the attacker adds a significant amount of collateral, they will receive substantial profits in the form of xToken rewards.

(3). The attacker can call the `redeem()` function to burn fToken and get baseToken.

After completing the second step, the system's collateral ratio will reach the stable warning line. At this point, burning fToken through redemption will further increase the collateral ratio.

(4). The attacker can call the `redeem()` function to burn xToken and get baseToken.

By burning all the xToken held by the attacker and retrieving the baseToken, the attack does not directly destabilize the system. However, it does result in the theft of funds from all fToken holders (the attack will lower the fNAV value, directly affecting the value of fToken holders).

The four steps mentioned above are performed as atomic operations within the same attack function, and they can be looped to expand the impact of the attack.

```
// @audit located in Market.sol
function mintFToken(
    uint256 _baseIn,
    address _recipient,
    uint256 _minFTokenMinted
```

```
  ) external override nonReentrant cachePrice returns (uint256
_fTokenMinted) {
    ......

    if (fTokenMintInSystemStabilityModePaused) {
      uint256 _collateralRatio = _treasury.collateralRatio();
      require(_collateralRatio > marketConfig.stabilityRatio,
"fToken mint paused");

      // bound maximum amount of base token to mint fToken.
      if (_baseIn > _maxBaseInBeforeSystemStabilityMode) {
        _baseIn = _maxBaseInBeforeSystemStabilityMode;
      }
    }

    uint256 _amountWithoutFee = _deductFTokenMintFee(_baseIn,
fTokenMintFeeRatio, _maxBaseInBeforeSystemStabilityMode);

    IERC20Upgradeable(_baseToken).safeTransferFrom(msg.sender,
address(_treasury), _amountWithoutFee);
    (_fTokenMinted, ) = _treasury.mint(_amountWithoutFee,
_recipient, ITreasury.MintOption.FToken);
    require(_fTokenMinted >= _minFTokenMinted, "insufficient
fToken output");

    emit Mint(msg.sender, _recipient, _baseIn, _fTokenMinted,
0, _baseIn - _amountWithoutFee);
  }

function addBaseToken(
    uint256 _baseIn,
    address _recipient,
    uint256 _minXTokenMinted
  ) external override nonReentrant cachePrice returns (uint256
_xTokenMinted) {
    ......

    (uint256 _maxBaseInBeforeSystemStabilityMode, ) =
_treasury.maxMintableXTokenWithIncentive(
      _marketConfig.stabilityRatio,
      incentiveConfig.stabilityIncentiveRatio
    );

    // bound the amount of base token
    FeeRatio memory _ratio = xTokenMintFeeRatio;
    uint256 _feeRatio = uint256(int256(_ratio.defaultFeeRatio)
+ _ratio.extraFeeRatio);
    if (_baseIn * (PRECISION - _feeRatio) >
_maxBaseInBeforeSystemStabilityMode * PRECISION) {
      _baseIn = (_maxBaseInBeforeSystemStabilityMode *
PRECISION) / (PRECISION - _feeRatio);
    }

   ......
  }

function redeem(
    uint256 _fTokenIn,
    uint256 _xTokenIn,
    address _recipient,
    uint256 _minBaseOut
  ) external override nonReentrant cachePrice returns (uint256
_baseOut) {
    require(!redeemPaused, "redeem is paused");

    if (_fTokenIn == uint256(-1)) {
      _fTokenIn =
IERC20Upgradeable(fToken).balanceOf(msg.sender);
    }
    if (_xTokenIn == uint256(-1)) {
      _xTokenIn =
IERC20Upgradeable(xToken).balanceOf(msg.sender);
    }
    require(_fTokenIn > 0 || _xTokenIn > 0, "redeem zero
amount");
    require(_fTokenIn == 0 || _xTokenIn == 0, "only redeem
single side");
```

```
    ITreasury _treasury = ITreasury(treasury);
    MarketConfig memory _marketConfig = marketConfig;

    uint256 _feeRatio;
    if (_fTokenIn > 0) {
      (, uint256 _maxFTokenInBeforeSystemStabilityMode) =
_treasury.maxRedeemableFToken(_marketConfig.stabilityRatio);
      _feeRatio = _computeRedeemFeeRatio(_fTokenIn,
fTokenRedeemFeeRatio, _maxFTokenInBeforeSystemStabilityMode);
    } else {
      (, uint256 _maxXTokenInBeforeSystemStabilityMode) =
_treasury.maxRedeemableXToken(_marketConfig.stabilityRatio);

      if (xTokenRedeemInSystemStabilityModePaused) {
        uint256 _collateralRatio =
_treasury.collateralRatio();
        require(_collateralRatio >
_marketConfig.stabilityRatio, "xToken redeem paused");

        // bound maximum amount of xToken to redeem.
        if (_xTokenIn > _maxXTokenInBeforeSystemStabilityMode)
{
          _xTokenIn = _maxXTokenInBeforeSystemStabilityMode;
        }
      }
  ......
  }
```

Status

The profit obtained by attackers is related to the platform fees and incentive ratio charged by the system. Therefore, the team has decided to protect the system from harm by adjusting the appropriate system parameters in a way that prevents attackers from gaining any profit.

4.3.5 Potential flash loan attacks to seize the system's liquidation rewards.

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

Description

Similar to the attack method described in issue 4.3.4, attackers can exploit the system's liquidation rewards by manipulating the collateral ratio and utilizing the `liquidate()` function. When the collateral ratio drops to the user's redemption liquidation line, the protocol encourages fToken holders to redeem by deducting the Net Asset Value (NAV) of fTokens. The unredeemed fTokens subsidize the redemption of fTokens. At this point, fToken holders receive a corresponding proportion of rewards upon redemption. The attack process can be simplified as follows:

(1). The attacker initiates a flash loan to borrow a large number of baseTokens and mints fTokens by calling the `mintFToken()` function, thereby reducing the collateral ratio. The lower the collateral ratio within the allowed range, the higher the profits gained.

(2). The attacker invokes the `liquidate()` function to perform liquidation, earning profits and recouping the principal.

(3). The attacker calls the `redeem()` function to burn the remaining fTokens and obtain baseTokens, completing the attack.

Similar to issue 4.3.4, attackers can achieve their goal of stealing funds by adjusting relevant parameters.

Note: White-listed users invoking the `selfLiquidate()` function can also achieve the attack objective. However, since white-listed users are authorized by the administrator, the overall risk is manageable.

```
// @audit located in Market.sol
```

```
function mintFToken(
    uint256 _baseIn,
    address _recipient,
    uint256 _minFTokenMinted
  ) external override nonReentrant cachePrice returns (uint256
_fTokenMinted) {
    ......

    if (fTokenMintInSystemStabilityModePaused) {
      uint256 _collateralRatio = _treasury.collateralRatio();
      require(_collateralRatio > marketConfig.stabilityRatio,
"fToken mint paused");

      // bound maximum amount of base token to mint fToken.
      if (_baseIn > _maxBaseInBeforeSystemStabilityMode) {
        _baseIn = _maxBaseInBeforeSystemStabilityMode;
      }
    }

    uint256 _amountWithoutFee = _deductFTokenMintFee(_baseIn,
fTokenMintFeeRatio, _maxBaseInBeforeSystemStabilityMode);

    IERC20Upgradeable(_baseToken).safeTransferFrom(msg.sender,
address(_treasury), _amountWithoutFee);
    (_fTokenMinted, ) = _treasury.mint(_amountWithoutFee,
_recipient, ITreasury.MintOption.FToken);
    require(_fTokenMinted >= _minFTokenMinted, "insufficient
fToken output");

    emit Mint(msg.sender, _recipient, _baseIn, _fTokenMinted,
0, _baseIn - _amountWithoutFee);
  }

/// @inheritdoc IMarket
  function liquidate(
    uint256 _fTokenIn,
    address _recipient,
    uint256 _minBaseOut
  ) external override nonReentrant cachePrice returns (uint256
_baseOut) {
    require(!redeemPaused, "redeem is paused");

    ITreasury _treasury = ITreasury(treasury);
    uint256 _collateralRatio = _treasury.collateralRatio();

    MarketConfig memory _marketConfig = marketConfig;
    require(
      _marketConfig.recapRatio <= _collateralRatio &&
_collateralRatio < _marketConfig.liquidationRatio,
      "Not liquidation mode"
    );

    // bound the amount of fToken
    (, uint256 _maxFTokenLiquidatable) =
_treasury.maxLiquidatable(
      _marketConfig.liquidationRatio,
      incentiveConfig.liquidationIncentiveRatio
    );
    if (_fTokenIn > _maxFTokenLiquidatable) {
      _fTokenIn = _maxFTokenLiquidatable;
    }

    _baseOut = _treasury.liquidate(_fTokenIn,
incentiveConfig.liquidationIncentiveRatio, msg.sender);

    ......
  }
```

Status

The team has decided to prevent attackers from profiting by adjusting the system parameters, similar to the solution provided in issue 4.3.4.

4.3.6 The current code does not take into account the impact of platform fees on the actual principal amount deposited into the contract.

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

Description

After enabling the stability mechanism, the protocol does not allow users to mint fTokens without limitations. It calculates the maximum amount of `baseToken` that can be stored, satisfying the stability mechanism's warning line, denoted as `_maxBaseInBeforeSystemStabilityMode`. If the user intends to deposit more `baseToken` than this maximum value, it will be capped at that value. It is important to note that in the current code, after determining the maximum amount of `baseToken` that can be deposited, platform fees are deducted from it. Only the remaining funds, represented as `_amountWithoutFee`, are used to mint fTokens. However, the rest will be less than the maximum allowable amount `_maxBaseInBeforeSystemStabilityMode`. It leads to an insufficient actual principal amount being deposited.

```
// @audit located in Market.sol
function mintFToken(
    uint256 _baseIn,
    address _recipient,
    uint256 _minFTokenMinted
  ) external override nonReentrant cachePrice returns (uint256
_fTokenMinted) {
    ......
    ITreasury _treasury = ITreasury(treasury);

     //@audit The amount of baseToken that exactly meets the
stability mechanism's warning line is calculated without
considering platform fees.
```

```
    (uint256 _maxBaseInBeforeSystemStabilityMode, ) =
_treasury.maxMintableFToken(marketConfig.stabilityRatio);

    if (fTokenMintInSystemStabilityModePaused) {
      uint256 _collateralRatio = _treasury.collateralRatio();
      require(_collateralRatio > marketConfig.stabilityRatio,
"fToken mint paused");

      // bound maximum amount of base token to mint fToken.
      //@audit platform fees are not taken into account
      if (_baseIn > _maxBaseInBeforeSystemStabilityMode) {
        _baseIn = _maxBaseInBeforeSystemStabilityMode;
      }
    }

    //@audit After deducting platform fees, the actual amount
of `baseToken` used for minting fTokens will be lower than the
allowed maximum value.
    uint256 _amountWithoutFee = _deductFTokenMintFee(_baseIn,
fTokenMintFeeRatio, _maxBaseInBeforeSystemStabilityMode);

    IERC20Upgradeable(_baseToken).safeTransferFrom(msg.sender,
address(_treasury), _amountWithoutFee);
    (_fTokenMinted, ) = _treasury.mint(_amountWithoutFee,
_recipient, ITreasury.MintOption.FToken);

    ......
  }
```

Status

The team confirms that the design logic aligns with the requirements.

4.3.7 From a user's perspective, the functions `mintXToken()` and `addBaseToken()` may have overlapping functionalities under specific conditions.

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

Description

In the case where the system is not paused, users can use the `mintXToken()` function to mint xTokens without modifying the system's Net Asset Value (NAV). The `addBaseToken()` function, on the other hand, can only be called when the system's collateral ratio is higher than the system's recapping threshold and lower than the system's stability warning line. Under these specific conditions, users can mint xTokens by calling the `addBaseToken()` function. The difference is that calling the `addBaseToken()` function to mint xTokens will also change the NAV value of the contract. From the user's perspective, when the specific conditions mentioned above are met, both functions can be used to obtain xTokens. However, the advantage for users is that calling the `addBaseToken()` function allows them to receive corresponding incentives. Therefore, when the collateral ratio is higher than the system's recapping threshold and lower than the system's stability warning line, users would prefer to call the `addBaseToken()` function. In this case, the `mintXToken()` function becomes unnecessary. It is important to confirm whether the `mintXToken()` function is still applicable under the specific conditions mentioned above.

```
// @audit located in Market.sol
function mintXToken(
  uint256 _baseIn,
  address _recipient,
  uint256 _minXTokenMinted
) external override nonReentrant cachePrice returns (uint256
_xTokenMinted) {
  require(!mintPaused, "mint is paused");

  ......

  IERC20Upgradeable(_baseToken).safeTransferFrom(msg.sender,
address(_treasury), _amountWithoutFee);
  (, _xTokenMinted) = _treasury.mint(_amountWithoutFee,
_recipient, ITreasury.MintOption.XToken);
  require(_xTokenMinted >= _minXTokenMinted, "insufficient
xToken output");

  emit Mint(msg.sender, _recipient, _baseIn, 0, _xTokenMinted,
_baseIn - _amountWithoutFee);
}

/// @inheritdoc IMarket
function addBaseToken(
  uint256 _baseIn,
  address _recipient,
  uint256 _minXTokenMinted
) external override nonReentrant cachePrice returns (uint256
_xTokenMinted) {
  require(!mintPaused, "mint is paused");

  ITreasury _treasury = ITreasury(treasury);
  uint256 _collateralRatio = _treasury.collateralRatio();

  MarketConfig memory _marketConfig = marketConfig;
  require(
    _marketConfig.recapRatio <= _collateralRatio &&
_collateralRatio < _marketConfig.stabilityRatio,
    "Not system stability mode"
  );

  (uint256 _maxBaseInBeforeSystemStabilityMode, ) =
_treasury.maxMintableXTokenWithIncentive(
    _marketConfig.stabilityRatio,
    incentiveConfig.stabilityIncentiveRatio
  );
  ......

  _xTokenMinted = _treasury.addBaseToken(_baseIn,
incentiveConfig.stabilityIncentiveRatio, _recipient);
  require(_xTokenMinted >= _minXTokenMinted, "insufficient
xToken output");

  emit AddCollateral(msg.sender, _recipient, _baseIn,
_xTokenMinted);
}
```

Status

The team has confirmed the issue. In cases where the collateral ratio is slightly below the stability line, the reward obtained by adding collateral using the `addBaseToken()` function may be minimal, possibly insufficient to cover the transaction fees. In such situations, allowing users to deposit all of their xTokens through the `mintXToken()` function would be more beneficial for users. Therefore, taking into account the users' actual needs, the team has decided to maintain the current design.

4.3.8 Risk Discussion of the `redeem()` Function.

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

Description

Users can redeem baseTokens by using the `redeem()` function, which requires them to burn the fTokens or xTokens they hold. It's worth noting that when users redeem xTokens for principal, there is a parameter called `xTokenRedeemInSystemStabilityModePaused` that determines whether the system adopts the stability mechanism mode. Let's discuss the two scenarios:

(1). When the parameter `xTokenRedeemInSystemStabilityModePaused` is set to true:

In this scenario, the system is in the stability mechanism mode, and to prevent users from redeeming xTokens to a point where the collateral ratio falls below the stability mechanism warning line, the system does not allow users to redeem xTokens beyond the `_maxXTokenInBeforeSystemStabilityMode` value. Let's consider the following scenarios:

- Where the system's collateral ratio is slightly higher than the stability mechanism warning line and the user holds a significant amount of fTokens and xTokens, the user chooses to redeem all of their principal：

  If the user chooses to redeem xTokens first and then redeem fTokens (as the system does not allow simultaneous redemption of both), there is a possibility that the user may exceed the `_maxXTokenInBeforeSystemStabilityMode` limit when redeeming xTokens, resulting in a collateral ratio dropping to the stability mechanism warning line. Subsequently, when redeeming all fTokens, the collateral ratio would increase again. Finally, the user would be able to redeem the remaining xTokens. This multiple-step redemption process can indeed result in inefficiencies for the user in terms of gas costs and increased expenses.

  From the perspective of the overall user population, when the collateral ratio is slightly higher than the stability mechanism warning line, redeeming xTokens to retrieve the principal can introduce significant uncertainty. The system's restriction on users redeeming excessive xTokens means that when the collateral ratio drops to the stability mechanism warning line, users must wait for other users to mint new xTokens or burn fTokens before they can continue redeeming their principal. It can result in increased time costs and gas costs for users.

- If the current collateral ratio is already below the stability mechanism warning line：

> Users want to increase the collateral ratio and maintain system stability by adding collateral through the `addBaseToken()` function and minting xTokens. However, if the price of the collateral continues to decline and the collateral ratio falls below the stability mechanism warning line again, users may be unable to redeem their principal. This design may affect the motivation of users to actively participate in maintaining system stability and could raise concerns among users.

(2). When `xTokenRedeemInSystemStabilityModePaused == false`: In this case, the system does not restrict the amount of xTokens users can redeem, which can lead to more severe financial losses. Consider the following scenario: Suppose an attacker holds numerous xTokens, and when they redeem all of these xTokens, the collateral ratio falls below the stability mechanism warning line. The attacker can execute an attack using the following steps:

a). The attacker first calls the `redeem()` function to burn all the xTokens they hold and obtain the baseToken, assuming the quantity is `amount1`.

b). Due to the attacker's destruction of xTokens, the collateral ratio falls below the stable mechanism's warning line. At this point, the user immediately calls the `addBasetoken()` function to add collateral and deposits all the baseTokens obtained during redemption into the system. As a result, the attacker once again obtains xTokens, and due to the presence of system incentives, the newly minted xTokens have a higher value than the baseTokens used for collateral. The difference between the two values represents the profit gained by the attacker.

c). Once again, the attacker calls the `redeem()` function to burn all the newly minted xTokens, causing the system's collateral ratio to fall below the stable mechanism's warning line. This time, the gap is widening compared to the first occurrence.

d). The attacker continues to call the `addBasetoken()` function to repeatedly add collateral, following this pattern.

e). The aforementioned actions will cause the fNAV to decrease over time, and the funds held in fTokens will be depleted by the attacker.

The aforementioned steps are part of the same attack function and are performed as an atomic operation.

```
// @audit located in Market.sol
function redeem(
  uint256 _fTokenIn,
  uint256 _xTokenIn,
  address _recipient,
  uint256 _minBaseOut
) external override nonReentrant cachePrice returns (uint256
_baseOut) {
  require(!redeemPaused, "redeem is paused");

  if (_fTokenIn == uint256(-1)) {
    _fTokenIn =
IERC20Upgradeable(fToken).balanceOf(msg.sender);
  }
  if (_xTokenIn == uint256(-1)) {
    _xTokenIn =
IERC20Upgradeable(xToken).balanceOf(msg.sender);
  }
  require(_fTokenIn > 0 || _xTokenIn > 0, "redeem zero
amount");
  require(_fTokenIn == 0 || _xTokenIn == 0, "only redeem
single side");

  ITreasury _treasury = ITreasury(treasury);
  MarketConfig memory _marketConfig = marketConfig;

  uint256 _feeRatio;
  if (_fTokenIn > 0) {
    (, uint256 _maxFTokenInBeforeSystemStabilityMode) =
_treasury.maxRedeemableFToken(_marketConfig.stabilityRatio);
    _feeRatio = _computeFTokenRedeemFeeRatio(_fTokenIn,
fTokenRedeemFeeRatio, _maxFTokenInBeforeSystemStabilityMode);
  } else {
    (, uint256 _maxXTokenInBeforeSystemStabilityMode) =
_treasury.maxRedeemableXToken(_marketConfig.stabilityRatio);
```

```
    if (xTokenRedeemInSystemStabilityModePaused) {
      uint256 _collateralRatio = _treasury.collateralRatio();
      require(_collateralRatio > _marketConfig.stabilityRatio,
"xToken redeem paused");

      // bound maximum amount of xToken to redeem.
      if (_xTokenIn > _maxXTokenInBeforeSystemStabilityMode) {
        _xTokenIn = _maxXTokenInBeforeSystemStabilityMode;
      }
    }

    _feeRatio = _computeXTokenRedeemFeeRatio(_xTokenIn,
xTokenRedeemFeeRatio, _maxXTokenInBeforeSystemStabilityMode);
  }

  _baseOut = _treasury.redeem(_fTokenIn, _xTokenIn,
msg.sender);
  uint256 _balance =
IERC20Upgradeable(baseToken).balanceOf(address(this));
  // consider possible slippage
  if (_balance < _baseOut) {
    _baseOut = _balance;
  }

  uint256 _fee = (_baseOut * _feeRatio) / PRECISION;
  if (_fee > 0) {
    IERC20Upgradeable(baseToken).safeTransfer(platform, _fee);
    _baseOut = _baseOut - _fee;
  }
  require(_baseOut >= _minBaseOut, "insufficient base
output");

  IERC20Upgradeable(baseToken).safeTransfer(_recipient,
_baseOut);

  emit Redeem(msg.sender, _recipient, _fTokenIn, _xTokenIn,
_baseOut, _fee);
}
```

Status

The team has acknowledged this issue. In non-emergency situations, when the parameter `xTokenRedeemInSystemStabilityModePaused` is set to false, it will not prevent users from redeeming baseTokens using xTokens. On the other hand, similar to the solutions for issues 4.3.4 and 4.3.5, the team has adjusted the system parameters to prevent the attacker from profiting and thereby mitigate the attack.

4.3.9 Parameter `_amount` assignment error.

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

Description

The function `_transferBaseToken()` mistakenly assigns the value of parameter `_amount` to parameter `_balance` during the assignment.

```
function _transferBaseToken(uint256 _amount, address
_recipient) internal {
    address _baseToken = baseToken;
    uint256 _balance =
IERC20Upgradeable(_baseToken).balanceOf(address(this));
    if (_balance < _amount) {
      uint256 _diff = _amount - _balance;
      IAssetStrategy(strategy).withdrawToTreasury(_diff);
      strategyUnderlying = strategyUnderlying.sub(_diff);

      _balance =
IERC20Upgradeable(_baseToken).balanceOf(address(this));
      if (_amount > _balance) {
        //@audit the assignment is incorrect here.
        _balance = _balance;
      }
    }
    IERC20Upgradeable(_baseToken).safeTransfer(_recipient,
_amount);
  }
```

Suggestion

Modify the code as follows:

```
function _transferBaseToken(uint256 _amount, address
_recipient) internal {
    address _baseToken = baseToken;
    uint256 _balance =
IERC20Upgradeable(_baseToken).balanceOf(address(this));
    if (_balance < _amount) {
      uint256 _diff = _amount - _balance;
      IAssetStrategy(strategy).withdrawToTreasury(_diff);
      strategyUnderlying = strategyUnderlying.sub(_diff);

      _balance =
IERC20Upgradeable(_baseToken).balanceOf(address(this));
      if (_amount > _balance) {
        //@audit modify
        _amount = _balance;
      }
    }

    IERC20Upgradeable(_baseToken).safeTransfer(_recipient,
_amount);
  }
```

Status

The team has accepted our suggestion and fixed the issue in commit 7636830.

5. Conclusion

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

Disclaimer

APPENDIX

Vulnerability/Risk Level Classification

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