Security Audit Report

arUSD of Concentrator

by AladdinDAO

SECBIT

June 18, 2024

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). Building upon the legacy f(x) protocol, the AladdinDAO team introduced a more competitive stablecoin, fxUSD(or rUSD), which boasts advantages such as Built-in yield, Scalability, Stability, and Resilience. The arUSD is an innovative auto-compounding token built on top of rUSD, designed to revolutionize how DAOs and individual users manage and grow their savings. Similar to high-yield savings accounts, arUSD provides users with the opportunity to earn interest on their holdings. SECBIT Labs conducted an audit from May 27 to June 18, 2024, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the arUSD of Concentrator 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 Admin invocation of the `rebalance()` function may cause users to lose a portion of their principal.	Medium	Fixed
Design & Implementation	4.3.2 When the admin invokes the `harvest()` function to perform compounding operations, some base tokens may be left in the contract, resulting in users losing a portion of their earnings.	Info	Discussed
Design & Implementation	4.3.3 Discussion on the logic of the `previewDeposit()` and `previewRedeem()` functions.	Info	Fixed
Design & Implementation	4.3.4 A potential sandwich attack could result in the theft of rewards within the contract.	Low	Discussed
Design & Implementation	4.3.5 Discussion on the logic of the `redeem()` function for fund redemption.	Medium	Fixed
Design & Implementation	4.3.6 Discussion on the logic of the `rebalance()` function.	Low	Fixed
Design & Implementation	4.3.7 A potential sandwich attack could lead to users losing their principal.	Info	Discussed
Design & Implementation	4.3.8 Discussion on the logic of `mint()` and `withdraw()` functions.	Info	Discussed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the arUSD of Concentrator is shown below:

- Smart contract code
  - initial review commit
    - 9b0d098
  - final review commit
    - ff697b7

2.2 Contract List

The following content shows the contracts included in the arUSD of Concentrator, which the SECBIT team audits:

Name	Lines	Description
FxUSDCompounder.sol	297	The core contract for user asset deposits, withdrawals, and yield reinvestment, together with the `FxUSDStandardizedYieldBase` contract, forms a complete set of functionalities.
FxUSDCompounder4626.sol	121	Refactor the arUSD token according to the ERC-4626 standard interface.
FxUSDStandardizedYieldBase.sol	203	The auxiliary contract for user asset deposits, withdrawals, yield reinvestment, and the `FxUSDCompounder` contract, forms a complete set of functionalities.

3. Contract Analysis

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

3.1 Role Classification

Two key roles in the arUSD of Concentrator are the Governance Account and the Common Account.

- Governance Account
  - Description

    Contract Administrator
  - Authority

        - Reinvested funds after liquidation
        - Retrieve the user's earnings and deposit them into the rebalance pool
        - Transfer ownership
    - Method of Authorization

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

      Use the specified token to mint arUSD token shares and continuously earn compound interest.
    - Authority
        - Deposit / Redeem base token
    - Method of Authorization

      No authorization required

3.2 Functional Analysis

The arUSD of Concentrator is a tokenized version of Convex Finance's rUSD EtherFi stable pool vault. It provides a powerful platform to maximize returns and optimize users' yield strategies. 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:

FxUSDCompounder & FxUSDStandardizedYieldBase

Users can mint arUSD tokens through these contracts, and holders of arUSD tokens can enjoy the benefits of automatic compounding. The main function in this contract are as follows:

- `rebalance()`

  Convert liquidated base token to fxUSD token.
- `harvest()`

  Harvest pending rewards from the `FxUSDCompounder` contract.
- `deposit()`

  Mint amounts of shares by depositing base tokens.
- `redeem()`

  Redeem an amount of base tokens by burning some shares.

FxUSDCompounder4626

The arUSD token is encapsulated as a standard ERC4626 token through this contract. The main function in this contract are as follows:

- `deposit()`

  Mint ERC4626 tokens by depositing specified tokens.
- `redeem()`

  Redeem specified tokens by burning some ERC4626 tokens.
- `mint()`

  Mint specified `shares` ERC4626 token by depositing specified tokens.
- `withdraw()`

  Redeem specified tokens by burning some ERC4626 tokens.
- `wrap()`

  Wrap arUSD tokens to ERC4626 tokens directly.
- `unwrap()`

  Unwrap ERC4626 tokens to arUSD tokens.
- `redeemToBaseToken()`

  Burn exactly shares from the owner and sends assets of base token to the receiver.

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 Admin invocation of the `rebalance()` function may cause users to lose a portion of their principal.

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

Location

FxUSDCompounder.sol#L199

Description

After full liquidation occurs in the rebalance pool, the contract administrator (rebalancer role) can invoke the `rebalance()` function to reinvest. It means the principal (base token) obtained from the liquidation will be minted into fToken and re-deposited into the rebalance pool. According to the current code logic, the `rebalance()` function uses the `IFxUSD(cachedFxUSD).mint()` function to mint the retrieved `cachedTotalBaseToken` amount of base tokens into fxusd tokens. However, in practice, the `cachedTotalBaseToken` amount of base tokens in the contract might not be fully utilized. This is because the `mint()` function checks the protocol's minimum collateral ratio to ensure that minting fTokens does not pose a security risk to the protocol. In such cases, the excess base tokens will remain in the `FxUSDCompounder` contract. Yet, the `rebalance()` function directly sets the parameter `totalPendingBaseToken` to 0, assuming all base tokens in the contract have been converted into fTokens and deposited into the rebalance pool, which clearly does not align with the actual situation.

In practice, the base token rewards retrieved from the rebalance pool primarily come from the principal of users who have been liquidated. If the `FxUSDCompounder` contract leaves some base tokens unprocessed, it can likely result in the loss of user principal. Additionally, if a liquidation occurs in the rebalance pool, and as long as the fxusd protocol's users mint fxusd (fToken) after the rebalance pool liquidation is completed (which causes the collateral ratio of the fxusd protocol to decrease), the principal obtained from the rebalance pool liquidation used entirely to mint fTokens might not be fully utilized due to collateral ratio limitations. At this point, invoking the `rebalance()` function of the `FxUSDCompounder` contract can easily result in some base tokens being left behind in the contract.

```
function mintFToken(
  uint256 _baseIn,
  address _recipient,
  uint256 _minFTokenMinted
) external override nonReentrant returns (uint256 _fTokenMinted) {
  if (mintPaused()) revert ErrorMintPaused();

  ......

  if (fTokenMintPausedInStabilityMode()) {
    uint256 _collateralRatio = IFxTreasuryV2(treasury).collateralRatio();
```

```
    if (_collateralRatio <= _stabilityRatio) revert
ErrorFTokenMintPausedInStabilityMode();

    // @audit check the min mortgage rate
    // bound maximum amount of base token to mint fToken.
    if (_baseIn > _maxBaseInBeforeSystemStabilityMode) {
      _baseIn = _maxBaseInBeforeSystemStabilityMode;
    }
  }
  ......
}
```

```
function rebalance(uint256 minFxUSD) external onlyRole(REBALANCER_ROLE) returns
(uint256 fxUSDOut) {
  address cachedVault = vault;
  uint256 cachedTotalBaseToken = totalPendingBaseToken;
  uint256 cachedTotalDepositedFxUSD = totalDepositedFxUSD;
  uint256 currentTotalFxUSD =
IFxShareableRebalancePool(pool).balanceOf(cachedVault);
  if (!_hasLiquidation(cachedTotalBaseToken, currentTotalFxUSD,
cachedTotalDepositedFxUSD)) {
    revert ErrNotLiquidatedBefore();
  }

  // claim pending base token first.
  IStakingProxyRebalancePool(cachedVault).getReward();

  address cachedBaseToken = baseToken;
  address cachedFxUSD = yieldToken;
  cachedTotalBaseToken =
IERC20Upgradeable(cachedBaseToken).balanceOf(address(this));

  _doApprove(cachedBaseToken, cachedFxUSD, cachedTotalBaseToken);

  // @audit We expect to mint all cachedTotalBaseToken amount of base tokens
into fxUSD. However, in practice, this may not be possible due to minimum
collateral ratio restrictions, leaving some base tokens unused.
  fxUSDOut = IFxUSD(cachedFxUSD).mint(cachedBaseToken, cachedTotalBaseToken,
address(this), minFxUSD);
  if (
    fxUSDOut <
    ((cachedTotalDepositedFxUSD - currentTotalFxUSD) * (RATE_PRECISION +
minRebalanceProfit)) / RATE_PRECISION
  ) {
    revert ErrInsufficientRebalancedFxUSD();
  }
  IStakingProxyRebalancePool(cachedVault).depositFxUsd(fxUSDOut);

  totalDepositedFxUSD = currentTotalFxUSD + fxUSDOut;
```

```
    // @audit In practice, due to minimum collateral ratio restrictions,
  cachedTotalBaseToken may not be fully utilized, so it will not be zero.
    totalPendingBaseToken = 0;

    emit Rebalance(_msgSender(), cachedTotalBaseToken, fxUSDOut);
  }
```

Status

The development team has confirmed this issue and has fixed it in commit be3fbd1.

4.3.2 When the admin invokes the `harvest()` function to perform compounding operations, some base tokens may be left in the contract, resulting in users losing a portion of their earnings.

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

Location

FxUSDCompounder.sol#L257

Description

Similar to the issue 4.3.1 described, the `harvest()` function may also leave some base tokens unprocessed when minting fTokens with the retrieved rewards (especially when minting fTokens near the collateral ratio stability line of the fxusd protocol, which can result in leftover base tokens in the contract). This can potentially lead to users losing part of their earnings.

```
  function harvest(
    address receiver,
    uint256 minBaseOut,
    uint256 minFxUSD
  ) external override onlyHarvester returns (uint256 baseOut, uint256 fxUSDOut) {
    ......
    // if liquidated, do nothing and return
    // otherwise, convert to FxUSD and distribute bounty
    uint256 cachedTotalBaseToken = totalPendingBaseToken;
    uint256 cachedTotalDepositedFxUSD = totalDepositedFxUSD;
    if (
      _hasLiquidation(
        cachedTotalBaseToken,
        IFxShareableRebalancePool(pool).balanceOf(cachedVault),
        cachedTotalDepositedFxUSD
      )
    ) {
```

```
    // @audit When the rebalance pool undergoes liquidation, the funds are not
reinvested but are retained within the contract.
    totalPendingBaseToken = cachedTotalBaseToken + baseOut;
    emit Harvest(_msgSender(), baseOut, 0, 0, 0);
  } else {
    // use base token as bounty
    uint256 expense = (getExpenseRatio() * baseOut) / RATE_PRECISION;
    if (expense > 0) {
      IERC20Upgradeable(cachedBaseToken).safeTransfer(treasury, expense);
    }
    uint256 bounty = (getHarvesterRatio() * baseOut) / RATE_PRECISION;
    if (bounty > 0) {
      IERC20Upgradeable(cachedBaseToken).safeTransfer(receiver, bounty);
    }

    address cachedFxUSD = yieldToken;
    uint256 amountBaseToken = baseOut - expense - bounty;
    _doApprove(cachedBaseToken, cachedFxUSD, amountBaseToken);

    // @audit There is still a possibility that some base tokens remain unused
and stay in the contract (as minting fTokens lowers the collateral ratio,
leaving extra base tokens in the contract).
    fxUSDOut = IFxUSD(cachedFxUSD).mint(cachedBaseToken, amountBaseToken,
address(this), 0);
    if (fxUSDOut < minFxUSD) revert ErrInsufficientHarvestedFxUSD();

    // already approved in `initialize` function.
    IStakingProxyRebalancePool(cachedVault).depositFxUsd(fxUSDOut);
    emit Harvest(_msgSender(), baseOut, expense, bounty, fxUSDOut);

    totalDepositedFxUSD = cachedTotalDepositedFxUSD + fxUSDOut;
  }
}
```

Status

The development team has acknowledged this issue. They explained as follows: “It is possible that there are some leftover base tokens. But this would also leave a way to manipulate (increase) the NAV of the compounder. We are fine with that.”

4.3.3 Discussion on the logic of the `previewDeposit()` and `previewRedeem()` functions.

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

Location

FxUSDStandardizedYieldBase.sol#L142-L189

Description

From the actual operation of the current fxusd protocol, it appears that the same fxusd contract may have more than one type of base token. For example, the contract at the address 0x65D72AA corresponds to the base tokens weETH and ezETH, each with different prices. However, the current `previewDeposit()` and `previewRedeem()` functions do not differentiate between these base tokens; instead, they use a fixed oracle to read price data. Suppose the oracle is set to return the price of weETH; in that case, a user mistakenly using it to determine the number of ezETH tokens required to receive a specified amount of arusd tokens might receive incorrect data. In such a scenario, users could obtain erroneous information.

```
function previewDeposit(address tokenIn, uint256 amountTokenToDeposit)
  external
  view
  override
  returns (uint256 amountSharesOut)
{
  if (!isValidTokenIn(tokenIn)) revert ErrInvalidTokenIn();

  uint256 amountFxUSD;
  if (tokenIn == yieldToken) {
    amountFxUSD = amountTokenToDeposit;
  } else {
    address _treasury = IFxShareableRebalancePool(pool).treasury();
    address oracle = IFxTreasuryV2(_treasury).priceOracle();
    (, , uint256 price, ) = IFxPriceOracleV2(oracle).getPrice();
    amountTokenToDeposit =
IFxTreasuryV2(_treasury).getUnderlyingValue(amountTokenToDeposit);
    amountFxUSD = (amountTokenToDeposit * price) / PRECISION;
  }

  uint256 _totalSupply = totalSupply();
  if (_totalSupply == 0) {
    amountSharesOut = amountFxUSD;
  } else {
    amountSharesOut = (amountFxUSD * _totalSupply) / totalDepositedFxUSD;
  }
}

/// @inheritdoc IStandardizedYield
/// @dev This function use lots of gas, not recommended to use on chain.
/// This function won't return correct data when there is a liquidation.
function previewRedeem(address tokenOut, uint256 amountSharesToRedeem)
  external
  view
  override
  returns (uint256 amountTokenOut)
```

```
{
  if (!isValidTokenOut(tokenOut)) revert ErrInvalidTokenOut();

  amountTokenOut = (amountSharesToRedeem * totalDepositedFxUSD) / totalSupply();
  // tokenOut is fxUSD or baseToken
  if (tokenOut != yieldToken) {
    address _treasury = IFxShareableRebalancePool(pool).treasury();
    address oracle = IFxTreasuryV2(_treasury).priceOracle();
    (, , , uint256 price) = IFxPriceOracleV2(oracle).getPrice();
    amountTokenOut = (amountTokenOut * PRECISION) / price;
    amountTokenOut = IFxTreasuryV2(_treasury).getWrapppedValue(amountTokenOut);
  }
}


function isValidTokenIn(address token) public view override returns (bool) {
  address fxUSD = yieldToken;
  if (token == fxUSD) return true;
  (address _fToken, , , , ) = FxUSD(fxUSD).markets(token);
  return _fToken != address(0);
}
```

Status

Before this issue was submitted, the development team had already identified and resolved it in commit a27e93e. As for the `previewRedeem()` function, since the protocol only allows redemption of a specific base token, the current logic is correct.

4.3.4 A potential sandwich attack could result in the theft of rewards within the contract.

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

Location

FxUSDCompounder.sol#L205-L266

Description

The current `FxUSDCompounder` contract distributes rewards in a lump-sum manner. The administrator (harvester) calls the `harvest()` function to collect rewards from the rebalance pool, convert them to base tokens, and then reinvest them back into the rebalance pool. This design increases the number of base tokens corresponding to each unit of arUSD tokens without changing the total number of arUSD tokens (i.e., the shares held by users). However, this approach can lead to sandwich attacks.

Attack Scenario:

An attacker, upon observing a transaction where the harvester calls the `harvest()` function, constructs a sequence of function calls (`deposit()` ==> `harvest()` ==> `redeem()`) within the same block to steal the protocol's earnings. Here's an illustrative example with simplified calculations (excluding transaction fees):

Initial Conditions:

- Rebalance pool has not been liquidated.
- Total fxusd tokens deposited via `FxUSDCompounder` in the rebalance pool: `totalDepositedFxUSD = 10000`
- Total arUSD tokens held by users: `_totalSupply = 8000`

Step 1: Attacker Deposits Funds

- Attacker calls `deposit()` to deposit 100,000 fxusd tokens. The attacker receives arUSD tokens calculated as follows:

$$amountSharesOut = \frac{(amountFxUSD \times \_totalSupply)}{\_totalFxUSD} = \frac{100000 \times 8000}{10000} = 80000 \quad (1)$$

- Total arUSD tokens after the attacker's deposit:

$$\_totalSupply = 8000 + 80000 = 88000 \quad (2)$$

- Total fxusd tokens in the rebalance pool after deposit:

$$totalDepositedFxUSD = 100000 + 10000 = 110000 \quad (3)$$

Step 2: Harvester Calls `harvest()`

- Harvester collects rewards (e.g., 1000 fxusd tokens) and reinvests them. Total fxusd tokens in the rebalance pool after reinvestment:

$$totalDepositedFxUSD = 110000 + 1000 = 111000 \quad (4)$$

Step 3: Attacker Redeems Funds

- The attacker calls `redeem()` to withdraw their principal. The amount of fxusd tokens the attacker can redeem is calculated as:

$$\begin{aligned} amountFxUSD &= \frac{(amountSharesToRedeem \times cachedTotalDepositedFxUSD)}{cachedTotalSupply} \\ &= \frac{80000 \times 111000}{88000} \\ &\doteq 100909 \end{aligned} \quad (5)$$

- The attacker's profit is:

$$100909 - 100000 = 909 \text{ fxusd tokens} \quad (6)$$

This attack scenario demonstrates a risk when the rebalance pool has not been liquidated. If the fxusd protocol meets liquidation conditions, an attacker could deposit a large amount of fTokens before the rebalance pool calls `liquidate()`, wait for the administrator to call `rebalance()` or `harvest()`, and then redeem their principal to gain excessive profits, harming ordinary users' earnings.

```
function harvest(
  address receiver,
  uint256 minBaseOut,
  uint256 minFxUSD
) external override onlyHarvester returns (uint256 baseOut, uint256 fxUSDOut) {
  ......
  // if liquidated, do nothing and return
  // otherwise, convert to FxUSD and distribute bounty
  uint256 cachedTotalBaseToken = totalPendingBaseToken;
  uint256 cachedTotalDepositedFxUSD = totalDepositedFxUSD;
  if (
    _hasLiquidation(
      cachedTotalBaseToken,
      IFxShareableRebalancePool(pool).balanceOf(cachedVault),
      cachedTotalDepositedFxUSD
    )
  ) {
    totalPendingBaseToken = cachedTotalBaseToken + baseOut;
    emit Harvest(_msgSender(), baseOut, 0, 0, 0);
  } else {
    // use base token as bounty
    uint256 expense = (getExpenseRatio() * baseOut) / RATE_PRECISION;
    if (expense > 0) {
      IERC20Upgradeable(cachedBaseToken).safeTransfer(treasury, expense);
    }
    uint256 bounty = (getHarvesterRatio() * baseOut) / RATE_PRECISION;
    if (bounty > 0) {
      IERC20Upgradeable(cachedBaseToken).safeTransfer(receiver, bounty);
    }

    address cachedFxUSD = yieldToken;
    uint256 amountBaseToken = baseOut - expense - bounty;
    _doApprove(cachedBaseToken, cachedFxUSD, amountBaseToken);
    fxUSDOut = IFxUSD(cachedFxUSD).mint(cachedBaseToken, amountBaseToken,
address(this), 0);
    if (fxUSDOut < minFxUSD) revert ErrInsufficientHarvestedFxUSD();

    // already approved in `initialize` function.
    IStakingProxyRebalancePool(cachedVault).depositFxUsd(fxUSDOut);
    emit Harvest(_msgSender(), baseOut, expense, bounty, fxUSDOut);

    // @audit calculate total fxusd deposited into rebalance pool
    totalDepositedFxUSD = cachedTotalDepositedFxUSD + fxUSDOut;
  }
```

```
}

/// @inheritdoc FxUSDStandardizedYieldBase
  function _deposit(address tokenIn, uint256 amountDeposited)
  internal
  virtual
  override
  returns (uint256 amountSharesOut)
{
  address cachedFxUSD = yieldToken;
  uint256 amountFxUSD;
  if (tokenIn == cachedFxUSD) {
    amountFxUSD = amountDeposited;
  } else {
    _doApprove(tokenIn, cachedFxUSD, amountDeposited);
    amountFxUSD = IFxUSD(cachedFxUSD).mint(tokenIn, amountDeposited,
address(this), 0);
  }

  // deposit into Convex, already approved in `initialize` function.
  IStakingProxyRebalancePool(vault).depositFxUsd(amountFxUSD);

  // convert to pool share
  uint256 _totalSupply = totalSupply();
  uint256 _totalFxUSD = totalDepositedFxUSD;
  if (_totalFxUSD == 0) {
    amountSharesOut = amountFxUSD;
  } else {
    // @audit get the shares
    amountSharesOut = (amountFxUSD * _totalSupply) / _totalFxUSD;
  }

  // @audit calculate total fxusd deposited into rebalance pool
  totalDepositedFxUSD = _totalFxUSD + amountFxUSD;
}

function _redeem(
  address receiver,
  address tokenOut,
  uint256 amountSharesToRedeem
) internal virtual override returns (uint256 amountTokenOut) {
  ......
  // If has liquidation, can only redeem as baseToken
  // Otherwise, withdraw as FxUSD first
  if (_hasLiquidation(cachedTotalBaseToken, currentTotalFxUSD,
cachedTotalDepositedFxUSD)) {
    if (tokenOut != cachedBaseToken) revert ErrInvalidTokenOut();

    // claim pending base token first.
```

```
    IStakingProxyRebalancePool(cachedVault).getReward();

    // use current real FxUSD/BaseToken balance for calculation.
    cachedTotalBaseToken =
IERC20Upgradeable(cachedBaseToken).balanceOf(address(this));
    amountFxUSD = (currentTotalFxUSD * amountSharesToRedeem) /
cachedTotalSupply;
    amountBaseToken = (cachedTotalBaseToken * amountSharesToRedeem) /
cachedTotalSupply;

    // withdraw as base token, since it may be impossible to wrap to FxUSD.
    _withdrawAsBase(cachedVault, amountFxUSD);
    uint256 baseTokenDelta =
IERC20Upgradeable(cachedBaseToken).balanceOf(address(this)) -
cachedTotalBaseToken;

    totalPendingBaseToken = cachedTotalBaseToken - amountBaseToken;
    totalDepositedFxUSD = cachedTotalDepositedFxUSD - amountFxUSD;
    amountBaseToken += baseTokenDelta;
    amountFxUSD = 0;
  } else {
    // just in case someone donate FxUSD to this contract.
    if (currentTotalFxUSD > cachedTotalDepositedFxUSD) cachedTotalDepositedFxUSD
= currentTotalFxUSD;

    // @audit calculate the amount of fxUSD retrieved
    amountFxUSD = (amountSharesToRedeem * cachedTotalDepositedFxUSD) /
cachedTotalSupply;

    totalDepositedFxUSD = cachedTotalDepositedFxUSD - amountFxUSD;
    if (tokenOut == cachedFxUSD) {
      // It is very rare but possible that the corresponding market is under
collateral.
      // In such case, we cannot withdraw as FxUSD and can only withdraw as base
token.
      // But since it is very rare, we don't do anything about this for now and
let the call revert.
      // The user will choose to withdraw as base token instead.
        IStakingProxyRebalancePool(cachedVault).withdrawFxUsd(amountFxUSD);
    } else {
      amountBaseToken =
IERC20Upgradeable(cachedBaseToken).balanceOf(address(this));
      _withdrawAsBase(cachedVault, amountFxUSD);
      amountBaseToken =
IERC20Upgradeable(cachedBaseToken).balanceOf(address(this)) - amountBaseToken;
      amountFxUSD = 0;
    }
  }
```

```
    if (amountBaseToken > 0) {
      IERC20Upgradeable(cachedBaseToken).safeTransfer(receiver, amountBaseToken);
      amountTokenOut = amountBaseToken;
    }
    if (amountFxUSD > 0) {
      IERC20Upgradeable(cachedFxUSD).safeTransfer(receiver, amountFxUSD);
      amountTokenOut = amountFxUSD;
    }
  }
```

Status

The developer team acknowledged this issue. They decided not to change the current logic to maintain code simplicity and reduce gas costs. Instead, they plan to mitigate potential security risks by increasing the frequency of `harvest()` function calls. Additionally, suppose attackers deposit funds before the rebalance pool is liquidated. In that case, their principal will also be liquidated, and the timing of `rebalance()` and `harvest()` function calls cannot be precisely predicted by the attackers, making the attack highly risky and unlikely to yield significant profits.

4.3.5 Discussion on the logic of the `redeem()` function for fund redemption.

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

Location

FxUSDCompounder.sol#L190-L195

FxUSDCompounder.sol#L353

FxUSDCompounder.sol#L373

Description

Users can call the `redeem()` function to withdraw their deposited principal at any time. Consider the following scenario: a user calls the `redeem()` function to withdraw their principal after the rebalance pool has been liquidated. (Note that at this point, the administrator has not called the `rebalance()` or `harvest()` functions to handle the earnings and liquidated principal.) This leads to two problems:

1. Loss of FXN Token Rewards: When the rebalance pool contract is liquidated, and a user directly calls the `redeem()` function, the `_hasLiquidation()` branch will execute. This retrieves the liquidated principal and rewards from the rebalance pool. However, the current code only processes rewards in the form of base tokens (primarily the user's liquidated principal) and does not handle other token rewards, such as FXN tokens. This means that when a user redeems their principal, they lose the FXN token rewards accumulated from the last `harvest()` to the present. The administrator must call the `harvest()` function in advance to convert FXN tokens to base tokens.

2. Incorrect Update of `totalDepositedFxUSD`: The user's liquidated principal will be converted into base tokens and sent to the `FxUSDCompounder` contract. The user will retrieve base tokens proportionally from both the `FxUSDCompounder` and rebalance pool contracts. Here's an example with simplified data (excluding transaction fees):

Scenario:

- Before liquidation, the `FxUSDCompounder` contract has deposited 100 fxusd tokens into the rebalance pool via the convex vault contract, and the total arUSD token supply is 100.
- The rebalance pool undergoes liquidation, with 60 fxusd tokens liquidated, leaving 40 remaining in the convex vault.
- The liquidated 60 fxusd tokens are returned to the `FxUSDCompounder` contract as base token rewards.
- A user calls `redeem()` to withdraw 70 arUSD tokens.

Calculations:

- Retrieval from the Rebalance Pool:

$$\begin{aligned} amountFxUSD &= \frac{(currentTotalFxUSD \times amountSharesToRedeem)}{cachedTotalSupply} \\ &= \frac{40 \times 70}{100} \\ &= 28 \end{aligned} \tag{7}$$

  Remaining principal in the rebalance pool:

$$40 - 28 = 12 \text{ fxusd tokens} \tag{8}$$

- Retrieval from the `FxUSDCompounder` Contract:

$$\begin{aligned} amountBaseToken &= \frac{(cachedTotalBaseToken \times amountSharesToRedeem)}{cachedTotalSupply} \\ &= \frac{60 \times 70}{100} \\ &= 42 \end{aligned} \tag{9}$$

  Remaining funds in the `FxUSDCompounder` contract:

$$60 - 42 = 18 \text{ fxusd tokens} \tag{10}$$

  (For simplicity, assume base tokens have equivalent value to fxusd tokens.)

- Total Principal Value Received by User:

$$28 + 42 = 70 \text{ fxusd tokens} \tag{11}$$

  (This does not include additional rewards, and the user receives base tokens equivalent to 70 fxusd tokens, which is correct.)

- Updating `totalDepositedFxUSD`:

$$\begin{aligned} \text{totalDepositedFxUSD} &= \text{cachedTotalDepositedFxUSD} - \text{amountFxUSD} \\ &= 100 - 28 = 72 \end{aligned} \tag{12}$$

This value is incorrect since the actual remaining principal is only 12 fxusd tokens, as the `cachedTotalDepositedFxUSD` retains the value from before the rebalance pool liquidation and has not been updated.

When in liquidation mode, if the administrator calls the `rebalance()` function, it will check if the remaining base tokens can be converted back to fxusd tokens. The `FxUSDCompounder` contract has only 18 fxusd tokens' worth of base tokens. The following inequality will be observed:

```
fxUSDOut <
   ((cachedTotalDepositedFxUSD - currentTotalFxUSD) * (RATE_PRECISION +
minRebalanceProfit)) / RATE_PRECISION
```

Substituting the values:

$$18 < \frac{(72 - 12) \times 1.05}{1} = 63 \tag{13}$$

Since this inequality holds true, the `rebalance()` function will fail, leaving the protocol in liquidation mode and unable to continue normal deposits (users can still withdraw funds normally).

```
function rebalance(uint256 minFxUSD) external onlyRole(REBALANCER_ROLE) returns
(uint256 fxUSDOut) {
   ......

  _doApprove(cachedBaseToken, cachedFxUSD, cachedTotalBaseToken);
  fxUSDOut = IFxUSD(cachedFxUSD).mint(cachedBaseToken, cachedTotalBaseToken,
address(this), minFxUSD);
  if (
    fxUSDOut <
    ((cachedTotalDepositedFxUSD - currentTotalFxUSD) * (RATE_PRECISION +
minRebalanceProfit)) / RATE_PRECISION
  ) {
    revert ErrInsufficientRebalancedFxUSD();
  }
    ......
  }


function _redeem(
  address receiver,
  address tokenOut,
  uint256 amountSharesToRedeem
) internal virtual override returns (uint256 amountTokenOut) {
  address cachedFxUSD = yieldToken;
  address cachedBaseToken = baseToken;
  address cachedVault = vault;
```

```
  // `_burn` is called before this function call, so we add it back here.
  uint256 cachedTotalSupply = totalSupply() + amountSharesToRedeem;

  // @audit Although liquidation has already occurred in the rebalance pool, the
value here still retains the pre-liquidation value without timely correction.
  uint256 cachedTotalDepositedFxUSD = totalDepositedFxUSD;
  uint256 cachedTotalBaseToken = totalPendingBaseToken;

  uint256 currentTotalFxUSD =
IFxShareableRebalancePool(pool).balanceOf(cachedVault);
  uint256 amountFxUSD;
  uint256 amountBaseToken;
  // If has liquidation, can only redeem as baseToken
  // Otherwise, withdraw as FxUSD first
  if (_hasLiquidation(cachedTotalBaseToken, currentTotalFxUSD,
cachedTotalDepositedFxUSD)) {
    if (tokenOut != cachedBaseToken) revert ErrInvalidTokenOut();

    // claim pending base token first.
    // @audit Retrieve the liquidated principal and earnings.
    IStakingProxyRebalancePool(cachedVault).getReward();

    // use current real FxUSD/BaseToken balance for calculation.
    cachedTotalBaseToken =
IERC20Upgradeable(cachedBaseToken).balanceOf(address(this));
    amountFxUSD = (currentTotalFxUSD * amountSharesToRedeem) /
cachedTotalSupply;
    amountBaseToken = (cachedTotalBaseToken * amountSharesToRedeem) /
cachedTotalSupply;

    // withdraw as base token, since it may be impossible to wrap to FxUSD.
    _withdrawAsBase(cachedVault, amountFxUSD);
    uint256 baseTokenDelta =
IERC20Upgradeable(cachedBaseToken).balanceOf(address(this)) -
cachedTotalBaseToken;

    // @audit The unprocessed base tokens retained by this contract.
    totalPendingBaseToken = cachedTotalBaseToken - amountBaseToken;

    // @audit The total amount of fxUSD tokens deposited into the rebalance pool
through the FxUSDCompounder contract.
    totalDepositedFxUSD = cachedTotalDepositedFxUSD - amountFxUSD;
    amountBaseToken += baseTokenDelta;
    amountFxUSD = 0;
  } else {
    ......
}
```

Status

The developer team identified and resolved this issue before it was submitted in commit d8d7138.

4.3.6 Discussion on the logic of the `rebalance()` function.

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

Location

FxUSDCompounder.sol#L182

Description

The administrator (rebalancer role) can call the `rebalance()` function to convert the user's liquidated principal and earnings into fToken (fxusd) and re-deposit them into the rebalance pool contract. The `rebalance()` function internally calls the `cachedVault.getReward()` function, which directly retrieves the liquidated principal and earnings from the rebalance pool contract. However, it does not handle rewards in the form of tokens other than the base token (such as FXN tokens).

Consider the following scenario: after the rebalance pool undergoes liquidation, the administrator (rebalancer role) directly calls the `rebalance()` function to handle the liquidated funds. Two potential issues arise:

1. Insufficient Use of Available Funds: Since the `rebalance()` function does not process rewards other than the base token (FXN tokens), it does not convert these other types of rewards into base tokens. As a result, when the administrator calls the `rebalance()` function for reinvestment, not all available funds in the contract are utilized. This leads to a smaller-than-optimal number of fxusd tokens being minted (`fxUSDOut`), which might cause the following condition to be met:

```
if(
    fxUSDOut <
    ((cachedTotalDepositedFxUSD - currentTotalFxUSD) * (RATE_PRECISION +
minRebalanceProfit)) / RATE_PRECISION
  ){
    revert ErrInsufficientRebalancedFxUSD();
}
```

In this case, the administrator would not be able to successfully call the `rebalance()` function. (This issue can be resolved by calling the `harvest()` function before calling the `rebalance()` function to handle all available funds properly.)

2. Unrecorded Token Rewards: Suppose the administrator successfully calls the `rebalance()` function without handling other types of token rewards (like FXN tokens). In that case, these unprocessed token rewards will temporarily remain in the contract but will not be recorded. When users subsequently call the `redeem()` function, they will receive less principal and miss

out on these rewards. Additionally, if other types of token rewards accumulate excessively in the `FxUSDCompounder` contract (remaining unprocessed), attackers could exploit this situation through sandwich attacks to steal these rewards, similar to the attack method described in Issue 4.3.4.

```
function rebalance(uint256 minFxUSD) external onlyRole(REBALANCER_ROLE) returns
(uint256 fxUSDOut) {
  address cachedVault = vault;
  uint256 cachedTotalBaseToken = totalPendingBaseToken;
  uint256 cachedTotalDepositedFxUSD = totalDepositedFxUSD;
  uint256 currentTotalFxUSD =
IFxShareableRebalancePool(pool).balanceOf(cachedVault);
  if (!_hasLiquidation(cachedTotalBaseToken, currentTotalFxUSD,
cachedTotalDepositedFxUSD)) {
    revert ErrNotLiquidatedBefore();
  }

  // claim pending base token first.
  // @audit To withdraw the principal and earnings liquidated from this
contract.
  IStakingProxyRebalancePool(cachedVault).getReward();

  address cachedBaseToken = baseToken;
  address cachedFxUSD = yieldToken;
  cachedTotalBaseToken =
IERC20Upgradeable(cachedBaseToken).balanceOf(address(this));

  _doApprove(cachedBaseToken, cachedFxUSD, cachedTotalBaseToken);
  fxUSDOut = IFxUSD(cachedFxUSD).mint(cachedBaseToken, cachedTotalBaseToken,
address(this), minFxUSD);
  if (
    fxUSDOut <
    ((cachedTotalDepositedFxUSD - currentTotalFxUSD) * (RATE_PRECISION +
minRebalanceProfit)) / RATE_PRECISION
  ) {
    revert ErrInsufficientRebalancedFxUSD();
  }
  IStakingProxyRebalancePool(cachedVault).depositFxUsd(fxUSDOut);

  totalDepositedFxUSD = currentTotalFxUSD + fxUSDOut;
  totalPendingBaseToken = 0;

  emit Rebalance(_msgSender(), cachedTotalBaseToken, fxUSDOut);
}
```

Status

The development team has acknowledged this issue. To simplify the code and reduce complexity, the team decided to leave these additional rewards for the `harvest()` function to handle. This approach ensures that all token rewards, including non-base tokens, are eventually processed and converted into base tokens, thus minimizing potential risks and ensuring users' principal and rewards are appropriately managed.

4.3.7 A potential sandwich attack could lead to users losing their principal.

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

Location

FxUSDCompounder4626.sol#L135

FxUSDCompounder4626.sol#L153

Description

The f(x) protocol allows minting of fToken using the lowest price data from different price sources. To mitigate the risk of sandwich attacks, the minting function `mintFToken()` and the redemption function `redeemFToken()` provide a minimum return quantity parameter (similar to slippage protection). However, in practice, the code sets the minimum mint quantity to 0, thus bypassing the protection mechanism designed into the code and reintroducing risk.

Additionally, the `redeem()` function suffers from the same issue. Further inspection reveals that the `withdrawAsBase()` function provided by the Convex protocol also lacks a minimum return value check when retrieving base tokens. This opens up potential sandwich attack vulnerabilities for users directly interacting with the Convex protocol.

```
function deposit(uint256 assets, address receiver) public override returns
(uint256 shares) {
  // we are sure this is not fee on transfer token
  address cachedAsset = asset;
  IERC20Upgradeable(cachedAsset).safeTransferFrom(_msgSender(), address(this),
assets);

  // @audit Note that setting the minimum amount of arUSD tokens minted to 0 can
result in users receiving significantly less than expected in the event of a
sandwich attack.
  shares = IStandardizedYield(compounder).deposit(address(this), cachedAsset,
assets, 0);

  _mint(receiver, shares);

  emit Deposit(_msgSender(), receiver, assets, shares);
}
```

```
function redeem(
  uint256 shares,
  address receiver,
  address owner
) public override returns (uint256 assets) {
  address sender = _msgSender();
  if (sender != owner) {
    _spendAllowance(owner, sender, shares);
  }

  _burn(owner, shares);

  // @audit There is also no minimum withdrawal value check being performed.
  assets = IStandardizedYield(compounder).redeem(receiver, shares, asset, 0,
false);

  emit Withdraw(sender, receiver, owner, assets, shares);
}
```

Status

The development team confirmed that fxUSD tokens or rUSD tokens are used as assets, so operations like swaps are unnecessary, and slippage checks are not necessary.

4.3.8 Discussion on the logic of `mint()` and `withdraw()` functions.

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

Location

FxUSDCompounder4626.sol#L159-L63

Description

The function `convertToAssets()` uses the oracle's maximum price when estimating the amount of base tokens needed to obtain the desired shares of arUSD tokens (equivalent to ERC4626 tokens). This can result in the amount of the estimated assets being less than the actual base tokens required to obtain the desired shares. The `deposit()` function, however, uses the oracle's minimum price when depositing these base tokens, leading to the actual minted arUSD tokens (equivalent to ERC4626 tokens) being less than the expected shares. This discrepancy should be highlighted on the front-end interface.

Additionally, as discussed in Issue 4.3.7, the sandwich attack risk present in the `deposit()` function may cause the actual arUSD tokens received by users to be significantly lower than expected, resulting in a loss of principal.

The same issue exists in the `withdraw()` function.

```
/// @inheritdoc IERC4626Upgradeable
function mint(uint256 shares, address receiver) external override returns
(uint256 assets) {
  // It uses the highest price from the oracle.
  assets = convertToAssets(shares);

  // It uses the lowest price from the oracle.
  deposit(assets, receiver);
}

/// @inheritdoc IERC4626Upgradeable
function convertToAssets(uint256 shares) public view override returns (uint256
assets) {
  assets = IStandardizedYield(compounder).previewRedeem(asset, shares);
}

/// @inheritdoc IERC4626Upgradeable
function withdraw(
  uint256 assets,
  address receiver,
  address owner
  ) external override returns (uint256 shares) {
    // It uses the lowest price from the oracle.
    shares = convertToShares(assets);

    // It uses the highest price from the oracle.
    redeem(shares, receiver, owner);
}
```

Status

The development team confirmed that fxUSD tokens or rUSD tokens are used as asset assets, so operations like swaps are unnecessary, and slippage checks are not necessary.

5. Conclusion

After auditing and analyzing Concentrator's arUSD, 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