Security Audit Report

btcUSD of the f(x) Protocol

by AladdinDAO

SECBIT

April 19, 2024

1. Introduction

The AladdinDAO is a decentralized network that shifts 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). The audit of the supplementary aspects of the f(x) protocol primarily focuses on three key areas:

- Expansion of Asset Support: A series of oracles have been added to broaden the potential applications of the f(x) protocol and support a wider range of assets.
- Enhanced User Convenience: To improve user convenience, the protocol introduces routing functionality, allowing users to deposit additional assets directly.
- Adjustment for wBTC Asset: Concerning the wBTC asset, the f(x) protocol has made minor logic adjustments by modifying the fee strategy.

SECBIT Labs conducted an audit from March 11 to April 19, 2024, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**.

The assessment shows that the btcUSD, Converter, and Oracle of f(x) Protocol have no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising (see part 4 for details).

Type	Description	Level	Status
Design & Implementation	4.3.1 The potential inaccuracies in the price data provided by Chainlink oracles could impact the proper functioning of the protocol.	Low	Discussed
Design & Implementation	4.3.2 The price data obtained from the Curve protocol's price_oracle may lag behind the actual prices.	Low	Discussed
Design & Implementation	4.3.3 Potential flash loan attacks could manipulate the `price_oracle` values, further affecting the functionality of the f(x) protocol.	Low	Discussed
Design & Implementation	4.3.4 Using the ETH-USD price as the safePrice may not align with actual usage scenarios.	Info	Fixed
Design & Implementation	4.3.5 Attackers can manipulate fund exchanges by forging the `_pool` parameter.	Low	Discussed
Design & Implementation	4.3.6 Merging redundant shift operations can save gas.	Info	Discussed
Design & Implementation	4.3.7 When using the `convert()` function for asset exchange, checking the quantity of assets being exchanged is important to prevent sandwich attacks.	Info	Fixed
Gas optimization	4.3.8 The logic related to calculating the `_fundingRate` parameter could be removed.	Info	Fixed
Design & Implementation	4.3.9 Users can pre-call the `harvest()` function to claim rewards, bypassing the logic of sending rewards to the platform.	Info	Discussed
Design & Implementation	4.3.10 Using the `Action.None` parameter within the `harvest()` function to update the protocol's NAV value may result in excessive xToken fees being charged.	Medium	Fixed
Design & Implementation	4.3.11 Discussion on the parameter `chainlinkMessageExpiration` value.	Info	Discussed
Design & Implementation	4.3.12 Due to the absence of consideration for protocol fees, the return values of the `collateralRatio()` function and the `isUnderCollateral()` function may be inaccurate.	Low	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about btcUSD, Converter, and Oracle of f(x) Protocol is shown below:

- Smart contract code
  - initial review commit
    - eaa2570
    - 44135bc
  - final review commit
    - 1ff4abd
    - b0e1d7b

2.2 Contract List

The following content shows the contracts included in btcUSD, Converter, and Oracle of f(x) Protocol, which the SECBIT team audits:

Name	Lines	Description
FxCVXTwapOracle.sol	46	The oracle returns the price of CVX-USD.
FxEETHTwapOracle.sol	46	The oracle returns the price of eETH-USD.
FxEzETHTwapOracle.sol	50	The oracle returns the price of ezETH-USD.
FxLSDOracleBase.sol	37	An abstract contract for LSD oracle.
FxPxETHTwapOracle.sol	52	The oracle returns the price of pxETH-USD.
FxTwapOracleBase.sol	32	An abstract contract to get TWAP price.
CurveNGConverter.sol	87	The routing contract integrated with functionalities related to the Curve NG pool.
ETHLSDConverter.sol	242	The routing contract integrated with functionalities related to ETH LSD token.
GeneralTokenConverterStorage.sol	46	The auxiliary contract facilitating exchange functionalities in conjunction with Uniswap V3 contracts.
UniswapV3Converter.sol	71	The routing contract integrated with functionalities related to uniswap v3 pool.
WETHConverter.sol	37	Convert WETH to ETH and send to the recipient.
CrvUSDBorrowRateAdapter.sol	42	The abstract contract for adjusting the protocol's funding cost fees.
MarketWithFundingCost.sol	22	The market contract with funding cost.
TreasuryWithFundingCost.sol	106	The treasury contract with funding cost.
FxChainlinkTwapOracle.sol	113	The contract that handles price data from Chainlink and provides it to the f(x) protocol.
FxWBTCTwapOracle.sol	64	The contract provides price data for WBTC-USDC to the protocol.
MarketV2.sol	431	The core contract for minting and redeeming btcUSDs and xTokens.
TreasuryV2.sol	407	A contract to store the baseToken, where the core functions can only be called by the Market contract.

Notice: This audit specifically focuses on the new code introduced in the `MarketV2` *and* `TreasuryV2` *contracts.*

3. Contract Analysis

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

3.1 Role Classification

Two key roles in btcUSD, Converter, and Oracle of f(x) Protocol are the Governance Account and the Common Account.

- Governance Account
  - Description

    Contract Administrator
  - Authority
    - Update protocol parameter
    - Transfer ownership
  - Method of Authorization

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

    Mint btcUSD token and xBTC by utilizing authorized base token
  - Authority
    - Mint / Burn btcUSD token
    - Mint / Burn xToken
    - Deposit btcUSDs token to rebalance pool
  - Method of Authorization

    No authorization required

3.2 Functional Analysis

The f(x) protocol implements a decentralized quasi-stablecoin with high collateral utilization efficiency and leveraged contracts with low liquidation risks and no funding costs. The SECBIT team conducted a detailed audit of some of the contracts in the protocol. We can divide the critical functions of the contract into four parts:

FxCVXTwapOracle & FxEETHTwapOracle & FxEzETHTwapOracle & FxLSDOracleBase & FxPxETHTwapOracle & FxTwapOracleBase & FxChainlinkTwapOracle & FxWBTCTwapOracle

These contracts aggregate data from various price sources, further process it, and provide it for use by the f(x) protocol. The main function is as follows:

- `getPrice()`

This function compares data from various price sources and assesses the validity of the prices, which are ultimately used for the core functionalities of the f(x) protocol.

CurveNGConverter & ETHLSDConverter & GeneralTokenConverterStorage & UniswapV3Converter & WETHConverter

As an auxiliary contract of the f(x) protocol, this contract provides routing functionality, assisting users in exchanging other supported tokens into the protocol's base token. The main functions are as follows:

- `getTokenPair()`

  Return the input token and output token for the route.
- `queryConvert()`

  Query the output token amount according to the encoding.
- `convert()`

  Convert the input token to the output token according to the encoding.
- `withdrawFund()`

  Withdraw dust assets in this contract.

MarketWithFundingCost & MarketV2

The f(x) protocol adopts a new strategy, charging users holding xTokens in the form of reducing xToken net asset value (NAV), which ultimately converts into baseToken fees. The main function is as follows:

- `harvestFundingCost()`

  Harvest funding costs from the treasury. If we have harvest bounty, transfer it to the platform.

TreasuryWithFundingCost & TreasuryV2 & CrvUSDBorrowRateAdapter

The f(x) protocol adopts a new strategy, charging users holding xTokens in the form of reducing xToken net asset value (NAV), which ultimately converts into baseToken fees. The main functions are as follows:

- `harvestable()`

  This function calculates the current base token fees to be charged.
- `harvest()`

  Users distribute the base token fees the protocol should collect 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 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 The potential inaccuracies in the price data provided by Chainlink oracles could impact the proper functioning of the protocol.

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

Location

FxCVXTwapOracle.sol#L58

Description

The `getPrice()` function in this contract retrieves price data for CVX-USDC. Specifically, the function compares the price data from the Chainlink oracle with the indirectly obtained price data for CVX-USDC from the Curve protocol. When the difference between the two price data is less than 1%, the price data for CVX-USDC is considered safe and valid. At this point, the protocol can use the price data provided by Chainlink for CVX-USDC for mint/redeem operations. However, upon inspection of the CVX-USDC price information provided on the Chainlink oracle website, it is found that the error in the price data provided by this oracle is within 2%. This means that within a limited time frame (24 hours, i.e., Heartbeat), as long as the error in the price data for CVX-USDC does not exceed 2%, the oracle will not update the price data. The precision error of this data does not meet the strict 1% requirement set by the protocol. In this scenario, if the data provided by the Curve protocol is valid but its price data falls within the (1%, 2%) range, the `getPrice()` function will consider the data provided by Chainlink unsafe, thus disallowing the protocol from performing mint operations, which affects the regular operation of the protocol.

For example, let's assume that the current price of CVX-USDC obtained from Chainlink is 3.6. This means that within a day, as long as the price obtained from its price source falls within the range ($3.6 \times (1 - 2\%) = 3.528$, $3.6 \times (1 + 2\%) = 3.672$), Chainlink will not update the price data and will maintain the price at 3.6. Now, if the price data for CVX-USDC obtained from the Curve protocol is 3.65 (which exceeds the 1% threshold), the `getPrice()` function will consider the price data for CVX-USDC unsafe, thereby prohibiting mint operations in the f(x) protocol. In reality, the price data for CVX-USDC is valid.

On the other hand, let's consider the impact of the precision error of the parameter `CVX_USDCurvePrice` on the output of the `getPrice()` function. This parameter consists of two parts: the ETH-USDC price data obtained from Chainlink, `ETH_USDChainlinkPrice`, and the CVX-ETH price data obtained from the Curve protocol, `CVX_ETHPrice`. By checking the Chainlink official website, we can determine that the error in the price data for ETH-USDC is within 0.5%. Therefore, it can be inferred that if the price

data for CVX-ETH obtained from the Curve protocol fluctuates by more than 0.5% (99.5% * 99.5% = 99%), the precision error of `CVX_USDCurvePrice` obtained will exceed 1%, causing the protocol to fail to operate properly.

Similar issues may exist in the `FxEETHTwapOracle` contract.

```
/// @dev The value of maximum price deviation
uint256 internal constant MAX_PRICE_DEVIATION = 1e16; // 1%

function getPrice()
    external
    view
    override
    returns (
      bool isValid,
      uint256 safePrice,
      uint256 minUnsafePrice,
      uint256 maxUnsafePrice
    )
  {
    // @audit get cvx-usdc price from chainlink
    uint256 CVX_USDChainlinkPrice = _getChainlinkBaseTwapUSDPrice();
    uint256 ETH_USDChainlinkPrice = _getChainlinkETHTwapUSDPrice();
    uint256 CVX_ETHPrice = _getCurveTwapETHPrice();
    uint256 CVX_USDCurvePrice = (ETH_USDChainlinkPrice * CVX_ETHPrice) /
PRECISION;

    safePrice = CVX_USDChainlinkPrice;
    // @audit check price data
    isValid = _isPriceValid(CVX_USDChainlinkPrice, CVX_USDCurvePrice,
MAX_PRICE_DEVIATION);

    // @note If the price is valid, `minUnsafePrice` and `maxUnsafePrice`
should never be used.
    // It is safe to assign them with Chainlink ETH/USD price.
    minUnsafePrice = CVX_USDChainlinkPrice;
    maxUnsafePrice = CVX_USDChainlinkPrice;
    if (!isValid) {
      minUnsafePrice = Math.min(CVX_USDChainlinkPrice,
CVX_USDCurvePrice);
      maxUnsafePrice = Math.max(CVX_USDChainlinkPrice,
CVX_USDCurvePrice);
    }
}
```

Suggestion

Consider adding an interface to modify the value of the `MAX_PRICE_DEVIATION` parameter. This would allow adjustments based on the actual situation after deployment to avoid impacting the protocol's normal operation.

Status

After comprehensively evaluating security risks, the development team has decided to keep the current parameter settings unchanged.

4.3.2 The price data obtained from the Curve protocol's price_oracle may lag behind the actual prices.

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

Location

FxCVXTwapOracle.sol#L83

Description

The on-chain address for the CurvePool is 0xB576491F1E6e5E62f1d8F26062Ee822B40B0E0d4. It is observed that the `price_oracle` parameter is only updated when users invoke functions such as `exchange()`, `add_liquidity()`, `remove_liquidity()`, etc. Therefore, when the `FxCVXTwapOracle` contract directly reads the `price_oracle()` value, it is highly likely to retrieve lagging price data, which can directly impact the core functionalities of the f(x) protocol.

On the other hand, based on the current on-chain data for the Curve CVX-ETH pool, the pool exhibits good depth, active trading, and relatively short update intervals for the `price_oracle()` parameter, resulting in relatively accurate price data. It is important to note that if the liquidity of this pool decreases, it may lead to more severe price data lagging issues, subsequently affecting the f(x) protocol. (Shallower depth, fewer users, and the potential for manipulation of `price_oracle` parameter values)

```
/// @dev Internal function to return the ETH price of CVX.
function _getCurveTwapETHPrice() internal view returns (uint256) {
  // The first token is ETH
  uint256 price = ICurvePoolOracle(curvePool).price_oracle();
  return price;
}
```

```
def tweak_price(A_gamma: uint256[2],_xp: uint256[N_COINS], p_i: uint256,
new_D: uint256):
    price_oracle: uint256 = self.price_oracle
    last_prices: uint256 = self.last_prices
    price_scale: uint256 = self.price_scale
    last_prices_timestamp: uint256 = self.last_prices_timestamp
    p_new: uint256 = 0

    if last_prices_timestamp < block.timestamp:
        # MA update required
        ma_half_time: uint256 = self.ma_half_time
        alpha: uint256 = self.halfpow((block.timestamp -
last_prices_timestamp) * 10**18 / ma_half_time)
        //@audit update price_oracle
        price_oracle = (last_prices * (10**18 - alpha) + price_oracle *
alpha) / 10**18
        self.price_oracle = price_oracle
        self.last_prices_timestamp = block.timestamp

    D_unadjusted: uint256 = new_D  # Withdrawal methods know new D
already
    if new_D == 0:
        # We will need this a few times (35k gas)
        D_unadjusted = self.newton_D(A_gamma[0], A_gamma[1], _xp)

    if p_i > 0:
        last_prices = p_i

    else:
        # calculate real prices
        __xp: uint256[N_COINS] = _xp
        dx_price: uint256 = __xp[0] / 10**6
        __xp[0] += dx_price
        last_prices = price_scale * dx_price / (_xp[1] -
self.newton_y(A_gamma[0], A_gamma[1], __xp, D_unadjusted, 1))

    self.last_prices = last_prices
```

```
    ......
```

Status

The development team has acknowledged this issue. They will monitor the corresponding pool after the protocol is operational to address any potential risks.

4.3.3 Potential flash loan attacks could manipulate the `price_oracle` values, further affecting the functionality of the f(x) protocol.

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

Location

FxEETHTwapOracle.sol#L85

Description

The on-chain data indicates minimal pool usage, with an average call interval exceeding 1 hour. It provides attackers with ample time to manipulate the `price_oracle` value. The specific steps for the attack are as follows:

(1). The attacker utilizes a flash loan to acquire a significant amount of WETH. They then call the `exchange()` function under the weETH-WETH pool to exchange WETH for weETH. Due to the inadequate pool depth, this action significantly deviates the price data in the pool from its actual value. It's important to note that the erroneous price data recorded will not be immediately used to calculate the `ema(price_oracle)` value, but it will update the `ma_last_time` parameter to the latest timestamp `block.timestamp`.

(2). Subsequently, the attacker calls the `exchange()` function again to exchange the obtained weETH back into WETH, restoring the pool to its state before the flash loan. At this point, the price data in the pool remains significantly deviated (in the opposite direction), and this value will be recorded in the `last_prices_packed_new[0]` parameter of the contract. Notably, the erroneous price data obtained in the first step will not be used to calculate the `ema(price_oracle)` value; instead, it will be overwritten by the latest erroneous price data.

These two steps, conducted via a flash loan, ultimately modify the latest price data (with erroneous data).

```
function _getCurveTwapETHPrice() internal view returns (uint256) {
    // The first token is weETH, and the price already consider
weETH.rate()
    uint256 price = ICurvePoolOracle(curvePool).price_oracle(0);
    return (PRECISION * PRECISION) / price;
}
```

```
@external
@view
@nonreentrant('lock')
def price_oracle(i: uint256) -> uint256:
    return self._calc_moving_average(
        self.last_prices_packed[i],
        self.ma_exp_time,
        self.ma_last_time & (2**128 - 1)
)
```

(3). Due to the infrequent usage of the weETH-WETH pool, approximately ten minutes later, the impact of this erroneous price data on the `ema(price_oracle)` value will reach 50%. At this point, the `ema(price_oracle)` data has significantly compromised the security of the f(x) protocol.

```
@internal
@view
def _calc_moving_average(
    packed_value: uint256,
    averaging_window: uint256,
    ma_last_time: uint256
) -> uint256:

    last_spot_value: uint256 = packed_value & (2**128 - 1)
    last_ema_value: uint256 = (packed_value >> 128)

    if ma_last_time < block.timestamp:  # calculate new_ema_value and
return that.
        alpha: uint256 = self.exp(
            -convert(
                (block.timestamp - ma_last_time) * 10**18 /
averaging_window, int256
            )
        )
```

```
        // @audit Based on the parameters deployed on-chain, after ten
minutes, the impact of the `last_spot_value` price (which is the
erroneous price data recorded by the attacker in the second step) on the
exponential moving average (ema) has reached 50%.
        return (last_spot_value * (10**18 - alpha) + last_ema_value *
alpha) / 10**18

    return last_ema_value
```

In theory, pools with insufficient depth or low user usage under the Curve protocol can potentially have their `price_oracle` values manipulated.

Similar issues may exist in the `FxPxETHTwapOracle` contract.

Status

The development team has acknowledged this issue and decided to redesign the oracle in subsequent versions. They explain that mint operations will not be allowed if the price is manipulated. In such a scenario, attackers will be unable to profit.

4.3.4 Using the ETH-USD price as the safePrice may not align with actual usage scenarios.

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

Location

FxEzETHTwapOracle.sol#L67

Description

The function `getPrice()` provides price data directly impacting the core functionalities of the f(x) protocol, such as minting and redeeming fTokens. The parameter `safePrice` is used to mint fTokens/xTokens within the protocol. Currently, the code utilizes the ETH-USD price provided by Chainlink as the `safePrice` for minting fToken (fezETH)/xToken. However, this does not align with the actual scenario. Specifically, considering the current code uses ezETH as the baseToken, theoretically, the price of ezETH-USD should be used for minting fToken/xToken, rather than the ETH-USD price. Currently, the secondary market's ezETH/ETH price is approximately $1\,ezETH \approx 1.007\,ETH$. If the ETH-USD price is used to mint fezETH tokens, calculated at the current price of $1\,ETH \approx 3500\,USD$, users minting 1 ezETH for fezETH would receive approximately 24.5 fewer tokens compared to minting with

the ezETH-USD price. This impact is more pronounced for large holders (receiving fewer fTokens compared to minting with the ezETH-USD price), necessitating confirmation of whether this logic aligns with requirements.

```
/// @inheritdoc IFxPriceOracle
/// @dev The price is valid iff

///  1. |Chainlink_ETH_USD - RedStone_ezETH_USDT / ezETH_rate | /
Chainlink_ETH_USD < 1%

function getPrice()
  external
  view
  override
  returns (
    bool isValid,
    uint256 safePrice,
    uint256 minUnsafePrice,
    uint256 maxUnsafePrice
  )
{
  uint256 ETH_USDChainlinkPrice = _getChainlinkTwapUSDPrice();
  uint256 ETH_USDUniswapPrice = _getUniV3TwapUSDPrice();
  uint256 ezETH_WETHCurvePrice = _getCurveTwapETHPrice();

  // @audit Use the price of ETH_USD as the price for EzETH_USD.
  safePrice = ETH_USDChainlinkPrice;

  isValid =
    _isPriceValid(ETH_USDChainlinkPrice, ETH_USDUniswapPrice,
MAX_PRICE_DEVIATION) &&
    _isPriceValid(PRECISION, ezETH_WETHCurvePrice, MAX_PRICE_DEVIATION);

  // @note If the price is valid, `minUnsafePrice` and `maxUnsafePrice`
should never be used.
  minUnsafePrice = ETH_USDChainlinkPrice;
  maxUnsafePrice = ETH_USDChainlinkPrice;
  if (!isValid) {
    uint256 ezETH_USDCurvePrice = (ezETH_WETHCurvePrice *
ETH_USDChainlinkPrice) / PRECISION;
    minUnsafePrice = Math.min(ETH_USDChainlinkPrice,
Math.min(ETH_USDUniswapPrice, ezETH_USDCurvePrice));
    maxUnsafePrice = Math.max(ETH_USDChainlinkPrice,
Math.max(ETH_USDUniswapPrice, ezETH_USDCurvePrice));
```

```
    }
  }
```

Status

The development team explains that the treasury will take into account the rate of ezETH.

4.3.5 Attackers can manipulate fund exchanges by forging the `_pool` parameter.

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

Location

CurveNGConverter.sol#L27

CurveNGConverter.sol#L89

ETHLSDConverter.sol#L86 (Note: Pool legality is not checked for protocol < 5 and protocol == 6).

ETHLSDConverter.sol#L133

ETHLSDConverter.sol#L182

UniswapV3Converter.sol#L45

UniswapV3Converter.sol#L63

UniswapV3Converter.sol#L91

Description

The current code encodes user parameters, including `_poolType`, `_action`, `_pool`, `indexIn`, `indexOut`, etc. The `getTokenPair()` function both reads these parameters and validates their legality. However, it's worth noting that the `_pool` parameter may be forged. Based on the specific code of the `getTokenPair()` function, it appears that bypassing the checks in the code only requires the forged `_pool` address contract to have the corresponding `coins()` function.

```
/// @dev Internal function to get the address of pool.
/// @param encoding The route encoding.
function _getPool(uint256 encoding) internal pure returns (address) {
  // @audit get pool address (can forge)
```

```
    return address((encoding >> 10) &
1461501637330902918203684832716283019655932542975);
  }

  /// @inheritdoc ITokenConverter
  function getTokenPair(uint256 _encoding) public view override returns
  (address _tokenIn, address _tokenOut) {
    uint256 _poolType = _getPoolType(_encoding);
    require(_poolType == 12 || _poolType == 13, "unsupported poolType");
    uint256 _action = _getAction(_encoding);

    address _pool = _getPool(_encoding);

    _encoding >>= 10;
    uint256 indexIn = (_encoding >> 163) & 7;
    uint256 indexOut = (_encoding >> 166) & 7;
    if (_action == 0) {
        _tokenIn = ICurveStableSwapNG(_pool).coins(indexIn);
        _tokenOut = ICurveStableSwapNG(_pool).coins(indexOut);
      } else if (_action == 1) {
      _tokenIn = ICurveStableSwapNG(_pool).coins(indexIn);
      _tokenOut = _pool;
    } else if (_action == 2) {
      _tokenIn = _pool;
        _tokenOut = ICurveStableSwapNG(_pool).coins(indexOut);
    } else {
      revert("unsupported action");
    }
  }
```

In the scenario of forging the `_pool` address, the `queryConvert()` function can return any `_amountOut` value desired by the attacker. When using the `convert()` function for fund conversion operations, attackers can steal these funds by forging the `_pool` parameter.

It's essential to confirm whether the upper-level functions calling the `getTokenPair()`, `queryConvert()`, and `convert()` functions check the `_encoding` parameter or set relevant calling permissions.

```
  function queryConvert(uint256 _encoding, uint256 _amountIn) external view
  override returns (uint256 _amountOut) {
    // to validate the encoding
    // @audit can be satisfied by forging the parameter _encoding
    getTokenPair(_encoding);
    uint256 _poolType = _getPoolType(_encoding);
```

```
  uint256 _action = _getAction(_encoding);
  address _pool = _getPool(_encoding);

  _encoding >>= 10;
  uint256 _tokens = ((_encoding >> 160) & 7) + 1;
  uint256 indexIn = (_encoding >> 163) & 7;
  uint256 indexOut = (_encoding >> 166) & 7;
  if (_action == 0) {
    _amountOut = ICurveStableSwapNG(_pool).get_dy(int128(indexIn),
int128(indexOut), _amountIn);
    } else if (_action == 1) {
      if (_poolType == 12) {
      uint256[] memory amounts = new uint256[](_tokens);
      amounts[indexIn] = _amountIn;
      _amountOut = ICurveStableSwapNG(_pool).calc_token_amount(amounts,
true);
    } else {
      uint256[2] memory amounts;
      amounts[indexIn] = _amountIn;
      _amountOut =
ICurveStableSwapMetaNG(_pool).calc_token_amount(amounts, true);
    }
  } else {
    _amountOut =
ICurveStableSwapNG(_pool).calc_withdraw_one_coin(_amountIn,
int128(indexOut));
  }
}

/// @inheritdoc ITokenConverter
function convert(
  uint256 _encoding,
  uint256 _amountIn,
  address _recipient
) external payable override returns (uint256 _amountOut) {
  // this will also validate the encoding
  // @audit the authenticity of the pool address is not verified
  (address _tokenIn, ) = getTokenPair(_encoding);

  uint256 _poolType = _getPoolType(_encoding);
  uint256 _action = _getAction(_encoding);

  address _pool = _getPool(_encoding);

  // only swap and add liquidity need to wrap and approve
```

```
  if (_action < 2) {
    _wrapTokenIfNeeded(_tokenIn, _amountIn);
    _approve(_tokenIn, _pool, _amountIn);
  }

  _encoding >>= 10;
  uint256 _tokens = ((_encoding >> 160) & 7) + 1;
  uint256 indexIn = (_encoding >> 163) & 7;
  uint256 indexOut = (_encoding >> 166) & 7;
  if (_action == 0) {
    _amountOut = ICurveStableSwapNG(_pool).exchange(int128(indexIn),
int128(indexOut), _amountIn, 0, _recipient);
  } else if (_action == 1) {
    if (_poolType == 12) {
      uint256[] memory amounts = new uint256[](_tokens);
      amounts[indexIn] = _amountIn;
      _amountOut = ICurveStableSwapNG(_pool).add_liquidity(amounts, 0,
_recipient);
    } else {
      uint256[2] memory amounts;
      amounts[indexIn] = _amountIn;
      _amountOut = ICurveStableSwapMetaNG(_pool).add_liquidity(amounts,
0, _recipient);
    }
  } else {
    _amountOut =
ICurveStableSwapNG(_pool).remove_liquidity_one_coin(_amountIn,
int128(indexOut), 0, _recipient);
  }
}
```

Status

The development team has acknowledged this issue. Considering that an attack is only feasible under frontend hijacking, which is relatively challenging, the team has decided to temporarily use this contract. The contract will be replaced in the future.

4.3.6 Merging redundant shift operations can save gas.

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

Location

CurveNGConverter.sol#L29-L31

CurveNGConverter.sol#L54-L57

CurveNGConverter.sol#L97-L100

Description

The following code performs a bit shift operation on `_encoding` before reading the `indexIn`, `indexOut`, and other parameters. Combining the bit shift operation into a single step can save gas.

```
function getTokenPair(uint256 _encoding) public view override returns
(address _tokenIn, address _tokenOut) {
  uint256 _poolType = _getPoolType(_encoding);
  require(_poolType == 12 || _poolType == 13, "unsupported poolType");
  uint256 _action = _getAction(_encoding);
  address _pool = _getPool(_encoding);

  // @audit combining the operation can save gas
  _encoding >>= 10;
  uint256 indexIn = (_encoding >> 163) & 7;
  uint256 indexOut = (_encoding >> 166) & 7;
  if (_action == 0) {
    ......
}

function queryConvert(uint256 _encoding, uint256 _amountIn) external view
override returns (uint256 _amountOut) {
  // to validate the encoding
  getTokenPair(_encoding);
  uint256 _poolType = _getPoolType(_encoding);
  uint256 _action = _getAction(_encoding);
  address _pool = _getPool(_encoding);

  // @audit combining the operation can save gas
  _encoding >>= 10;
  uint256 _tokens = ((_encoding >> 160) & 7) + 1;
  uint256 indexIn = (_encoding >> 163) & 7;
  uint256 indexOut = (_encoding >> 166) & 7;
  if (_action == 0) {
     ......
}

function convert(
```

```
    uint256 _encoding,
    uint256 _amountIn,
    address _recipient
  ) external payable override returns (uint256 _amountOut) {
    // this will also validate the encoding
    (address _tokenIn, ) = getTokenPair(_encoding);
    uint256 _poolType = _getPoolType(_encoding);
    uint256 _action = _getAction(_encoding);
      address _pool = _getPool(_encoding);

    // only swap and add liquidity need to wrap and approve
    if (_action < 2) {
      _wrapTokenIfNeeded(_tokenIn, _amountIn);
      _approve(_tokenIn, _pool, _amountIn);
    }

    // @audit combining the operation can save gas
    _encoding >>= 10;
    uint256 _tokens = ((_encoding >> 160) & 7) + 1;
    uint256 indexIn = (_encoding >> 163) & 7;
    uint256 indexOut = (_encoding >> 166) & 7;
    if (_action == 0) {
      ......
  }
```

Status

The development team has confirmed the issue and decided to optimize it in subsequent versions.

4.3.7 When using the `convert()` function for asset exchange, checking the quantity of assets being exchanged is important to prevent sandwich attacks.

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

Location

CurveNGConverter.sol#L102

CurveNGConverter.sol#L107

CurveNGConverter.sol#L111

CurveNGConverter.sol#L114

ETHLSDConverter.sol#L198

Descriptions

The provided code cannot set a minimum received token quantity when using the `exchange()`, `add_liquidity()`, `add_liquidity()`, and `remove_liquidity_one_coin()` functions. In cases of significant fluctuations in token prices or sandwich attacks, users might receive a significantly lower quantity of exchanged assets than expected. Therefore, it's crucial for the upper-level code to utilize the `convert()` function to perform checks on the exchanged asset quantity to prevent sandwich attacks. Currently, the code in the facets folder includes checks on the exchanged asset quantity. It's necessary to verify whether the `convert()` function serves other purposes to prevent any oversights.

```
function convert(
    uint256 _encoding,
    uint256 _amountIn,
    address _recipient
  ) external payable override returns (uint256 _amountOut) {
    // this will also validate the encoding
    (address _tokenIn, ) = getTokenPair(_encoding);
    uint256 _poolType = _getPoolType(_encoding);
    uint256 _action = _getAction(_encoding);
    address _pool = _getPool(_encoding);

    // only swap and add liquidity need to wrap and approve
    if (_action < 2) {
      _wrapTokenIfNeeded(_tokenIn, _amountIn);
      _approve(_tokenIn, _pool, _amountIn);
    }

    _encoding >>= 10;
    uint256 _tokens = ((_encoding >> 160) & 7) + 1;
    uint256 indexIn = (_encoding >> 163) & 7;
    uint256 indexOut = (_encoding >> 166) & 7;
    if (_action == 0) {
      // @audit uncheck slippage
      _amountOut = ICurveStableSwapNG(_pool).exchange(int128(indexIn),
int128(indexOut), _amountIn, 0, _recipient);
    } else if (_action == 1) {
      if (_poolType == 12) {
        uint256[] memory amounts = new uint256[](_tokens);
```

```
            amounts[indexIn] = _amountIn;
            // @audit uncheck slippage
            _amountOut = ICurveStableSwapNG(_pool).add_liquidity(amounts, 0,
_recipient);
        } else {
            uint256[2] memory amounts;
            amounts[indexIn] = _amountIn;
            // @audit uncheck slippage
            _amountOut = ICurveStableSwapMetaNG(_pool).add_liquidity(amounts,
0, _recipient);
        }
      } else {
        // @audit uncheck slippage
        _amountOut =
ICurveStableSwapNG(_pool).remove_liquidity_one_coin(_amountIn,
int128(indexOut), 0, _recipient);
      }
    }
```

Status

The development team has confirmed that all upper-level functions utilizing the `convert()` function perform slippage checks.

4.3.8 The logic related to calculating the `_fundingRate` parameter could be removed.

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

Location

CrvUSDBorrowRateAdapter.sol#L95-L107

Descriptions

The internal function `_captureFundingRate()` is only used in the `harvest()` function, and the return value `_fundingRate` is not utilized. Therefore, it may be beneficial to remove the calculation logic for the `_fundingRate` parameter within the `_captureFundingRate()` function to reduce gas consumption.

```
```

```
/// @dev Internal function to calculate the funding rate since last
snapshot and take snapshot.
function _captureFundingRate() internal returns (uint256 _fundingRate) {
  BorrowRateSnapshot memory cachedBorrowRateSnapshot =
borrowRateSnapshot;

  uint256 newBorrowIndex = ICrvUSDAmm(amm).get_rate_mul();
  _fundingRate =
    ((newBorrowIndex - uint256(cachedBorrowRateSnapshot.borrowIndex)) *
PRECISION) /
    uint128(cachedBorrowRateSnapshot.borrowIndex);
  _fundingRate = (_fundingRate * fundingCostScale) / PRECISION;

  cachedBorrowRateSnapshot.borrowIndex = uint128(newBorrowIndex);
  cachedBorrowRateSnapshot.timestamp = uint128(block.timestamp);
  borrowRateSnapshot = cachedBorrowRateSnapshot;
}
```

Suggestion

The logic for calculating the `_fundingRate` parameter will be removed as follows:

```
/// @dev Internal function to calculate the funding rate since last
snapshot and take snapshot.
// @audit remove the return value
function _captureFundingRate() internal //returns (uint256 _fundingRate)
{
  BorrowRateSnapshot memory cachedBorrowRateSnapshot =
borrowRateSnapshot;

  uint256 newBorrowIndex = ICrvUSDAmm(amm).get_rate_mul();

 // @audit remove the following codes
 // _fundingRate =
 //  ((newBorrowIndex - uint256(cachedBorrowRateSnapshot.borrowIndex)) *
PRECISION) /
 //   uint128(cachedBorrowRateSnapshot.borrowIndex);
 // _fundingRate = (_fundingRate * fundingCostScale) / PRECISION;

  cachedBorrowRateSnapshot.borrowIndex = uint128(newBorrowIndex);
  cachedBorrowRateSnapshot.timestamp = uint128(block.timestamp);
  borrowRateSnapshot = cachedBorrowRateSnapshot;
}
```

Status

The development team has adopted our suggestion and fixed this issue in commit f7eeeed.

4.3.9 Users can pre-call the `harvest()` function to claim rewards, bypassing the logic of sending rewards to the platform.

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

Location

MarketWithFundingCost.sol#L25

Descriptions

When users call the `mint` / `redeem` functions for `fToken` / `xToken`, the `harvestFundingCost()` function identifier is invoked to distribute protocol income and update the total principal quantity `totalBaseToken` under the protocol. At this point, the reward `_harvestBounty` belonging to the caller will be sent to the platform. Note that the harvest() function is of type external, meaning any user can directly call this function and receive the corresponding `_harvestBounty` reward. Therefore, as long as users call the `harvest()` function before calling the `mintFToken()` / `mintXToken()` / `redeemFToken()` / `redeemXToken()` functions, they can claim this portion of the bounty reward, causing the platform to lose this portion of income. It is necessary to confirm whether this logic meets the requirements.

```
/// @dev Harvest funding cost from treasury. If we have harvest bounty,
transfer to caller.
modifier harvestFundingCost() {
  uint256 _balance =
IERC20Upgradeable(baseToken).balanceOf(address(this));
  IFxTreasuryV2(treasury).harvest();
  uint256 _bounty = IERC20Upgradeable(baseToken).balanceOf(address(this))
- _balance;
  if (_bounty > 0) {
    //@audit send rewards to platform
    IERC20Upgradeable(baseToken).safeTransfer(platform, _bounty);
  }
  _;
}
```

Status

The development team has confirmed that this logic meets the requirements.

4.3.10 Using the `Action.None` parameter within the `harvest()` function to update the protocol's NAV value may result in excessive xToken fees being charged.

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

Location

TreasuryWithFundingCost.sol#L74

Descriptions

The protocol charges users holding xTokens, which results in a decrease in the xNav of xTokens. This fee is eventually deducted from the principal amount deposited by users and converted into baseTokens. The following `harvest()` function is used to collect fees for xTokens. Note that when calculating the baseNav and xNav under the protocol, `Action.None` is used. It means that even if the current oracle price is invalid, users can still call the `harvest()` function to update the protocol fees, directly affecting the total principal amount deposited by users (`totalBaseToken`), potentially leading to a loss of principal for users (invalid prices causing the protocol to overcharge xToken fees).

From the formulas, it is difficult to directly determine whether an invalid price ultimately leads to an excessive or insufficient amount of baseTokens charged by the protocol. This determination needs to be made based on the specific parameters and quantities of tokens during protocol runtime. However, it can be confirmed that using the `Action.None` parameter in the `harvest()` function directly affects users' principal when the oracle price is invalid.

```
function harvest() external virtual override {
  // no need to harvest
  if (borrowRateSnapshot.timestamp == block.timestamp) return;

  // update leverage
  // @audit use Action.None to calculate state
  FxStableMath.SwapState memory _state = _loadSwapState(Action.None);
  // silently return when under collateral since `MarketWithFundingCost`
would call this in each action.
  if (_state.xNav == 0) return;
  _updateEMALeverageRatio(_state);
```

```
    uint256 _totalRewards = harvestable();

    // @audit deduct fee
    totalBaseToken -= getUnderlyingValue(_totalRewards);
    _captureFundingRate();

    _distributedHarvestedRewards(_totalRewards);
  }

  function harvestable() public view override returns (uint256) {
    // @audit use Action.None to calculate state
    FxStableMath.SwapState memory _state = _loadSwapState(Action.None);

    // usually the leverage should always >= 1.0, but if < 1.0, the
  function will revert.
    uint256 _leverage = _state.leverageRatio();
    uint256 _fundingRate = getFundingRate();
    // funding cost = (xToken Value * (leverage - 1) / leverage * funding
  rate * scale) / baseNav
    uint256 _fundingCost = ((_state.xNav * _state.xSupply * (_leverage -
  PRECISION)) / _leverage);
    _fundingCost = (_fundingCost * _fundingRate) / PRECISION;
    _fundingCost /= _state.baseNav;

    return getWrapppedValue(_fundingCost);
  }
```

Status

The development team has confirmed this issue and fixed it in commit f7eeeed.

4.3.11 Discussion on the value of the parameter `chainlinkMessageExpiration`.

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

Location

FxChainlinkTwapOracle.sol#L30

Descriptions

According to the explanation of the parameter `chainlinkMessageExpiration`: "This should at least be equal to Chainlink's heartbeat duration," it indicates that the parameter value may be set to greater than the heartbeat of Chainlink. However, according to the official documentation of Chainlink, the provided price data update interval must not exceed the heartbeat. Otherwise, an error may have occurred. Therefore, setting the `chainlinkMessageExpiration` parameter value to exceed the heartbeat time is meaningless in this contract. For example, for the `getLatest()` function, if the `chainlinkMessageExpiration` value exceeds the heartbeat value, then the `require` condition cannot effectively check the validity of the price data because even if the Chainlink quote has expired (exceeded the heartbeat time but is less than the `chainlinkMessageExpiration` time), the require condition may still be satisfied. Using this outdated price data could introduce risks in such cases.

```
/// @notice Maximum gap between an epoch start and a previous Chainlink
round for the round
///         to be used in TWAP calculation.
/// @dev This should at least be equal to Chainlink's heartbeat duration.
uint256 public immutable chainlinkMessageExpiration;

 function getLatest() external view override returns (uint256) {
  (, int256 answer, , uint256 updatedAt, ) =
AggregatorV3Interface(chainlinkAggregator).latestRoundData();
  // @audit check whether the price update time has expired
  require(updatedAt >= block.timestamp - chainlinkMessageExpiration,
"Stale price oracle");
  return uint256(answer).mul(_chainlinkPriceMultiplier);
}
```

Status

The development team explains that through observation of on-chain results, under normal circumstances, the interval between Chainlink quotes may occasionally exceed Chainlink's heartbeat duration. This could occur due to fluctuations in gas prices, leading to transaction packaging times being later than expected.

4.3.12 Due to the absence of consideration for protocol fees, the return values of the `collateralRatio()` function and the `isUnderCollateral()` function may be inaccurate.

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

Location

TreasuryV2.sol#L183

TreasuryV2.sol#L193

Descriptions

Due to the fees charged on xTokens under the new protocol, the amount of principal deposited by users decreases, further affecting the protocol's collateral ratio. The `collateralRatio()` function and the `isUnderCollateral()` function below do not consider the impact of protocol fees on the return values of these two functions, which may result in inaccurate return values.

For the `collateralRatio()` function, the failure to consider the uncollected fees leads to an overestimation of the `baseSupply` parameter, ultimately resulting in an overestimation of the collateral ratio returned by this function.

```
function collateralRatio() public view override returns (uint256) {
  FxStableMath.SwapState memory _state = _loadSwapState(Action.None);

  if (_state.baseSupply == 0) return PRECISION;
  if (_state.fSupply == 0) return PRECISION * PRECISION;

  return (_state.baseSupply * _state.baseNav) / _state.fSupply;
}

function isUnderCollateral() public view returns (bool) {
    FxStableMath.SwapState memory _state = _loadSwapState(Action.None);
    return _state.xNav == 0;
}
```

Status

The development team has confirmed this issue and fixed it in commit f7eeeed.

5. Conclusion

After auditing and analyzing btcUSD, Converter and Oracle 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