Security Audit Report

Twap And Spot Price Oracle

of f(x) Protocol

by AladdinDAO

SECBIT

May 14, 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). The f(x) protocol used the TWAP (Time-Weighted Average Price) as the system's price oracle, which was directly utilized in the system's minting and redemption logic. Due to the inherent lag of TWAP prices, it couldn't promptly respond to sharp market fluctuations. The new oracle solution comprehensively considers TWAP and spot prices from various sources, maximizing the likelihood of swiftly capturing real-time market prices and reducing the probability of arbitrage occurrences. SECBIT Labs conducted an audit from April 30 to May 14, 2024, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the Twap And Spot Price Oracle of f(x) Protocol has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising (see part 4 for details).

Type	Description	Level	Status
Design & Implementation	4.3.1 Discussion on the risk of using spot prices from multiple sources.	Low	Discussed
Design & Implementation	4.3.2 Discussion on the return value of the function `_getLSDUSDTwap()`.	Info	Discussed
Design & Implementation	4.3.3 Malicious attackers can disrupt normal redemption behavior by minting small amounts of xTokens to users.	Low	Discussed
Design & Implementation	4.3.4 Discussion on handling the scale scheme.	Info	Discussed
Design & Implementation	4.3.5 Discussion on the `balances` parameter passed to the function `calculateInvariant()`.	Info	Discussed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about Twap And Spot Price Oracle of f(x) Protocol is shown below:

- Smart contract code
    - review commit
        - 9a5842b

2.2 Contract List

The following content shows the contracts included in Twap And Spot Price Oracle of f(x) Protocol, which the SECBIT team audits:

Name	Lines	Description
StableMath.sol	63	Handling data under the Curve protocol to obtain spot prices.
TreasuryWithFundingCost.sol	106	The treasury contract with funding cost.
FxStableMath.sol	98	The core code for minting and redeeming fTokens and xTokens.
FxBTCDerivativeOracleBase.sol	67	Return the price of BTC derivatives.
FxEETHOracleV2.sol	25	The oracle returns the price of eETH-USD.
FxEzETHOracleV2.sol	30	The oracle returns the price of ezETH-USD.
FxFrxETHOracleV2.sol	23	The oracle returns the price of FrxETH-USD.
FxLSDOracleV2Base.sol	127	An abstract contract for LSD oracle.
FxSpotOracleBase.sol	81	An abstract contract for spot price oracle.
FxStETHOracleV2.sol	23	The oracle returns the price of stETH-USD.
FxWBTCOracleV2.sol	22	The oracle returns the price of WBTC-USD.
LeveragedTokenV2.sol	72	Responsible for the standard ERC20 contract for minting and burning xTokens.
TreasuryV2.sol	402	A contract to store the baseToken, where the core functions can only be called by the Market contract.
SpotPriceOracle.sol	237	Handle the spot price from various price sources.

Notice: This audit specifically focuses on the new code introduced in the `TreasuryWithFundingCost`*,* `LeveragedTokenV2and`*, 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 Twap And Spot Price 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 fToken and xToken by utilizing authorized base token
  - Authority
    - Mint / Burn fToken
    - Mint / Burn xToken
  - Method of Authorization

    No authorization required

3.2 Functional Analysis

The f(x) protocol implements a decentralized quasi-stablecoin with high collateral utilization efficiency. This audit primarily focuses on the oracle component of the f(x) protocol. To address the issue of lagging TWAP prices during market volatility, the new oracle combines TWAP prices with spot prices to more rapidly reflect real-time market prices. 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:

FxBTCDerivativeOracleBase & FxEETHOracleV2 & FxEzETHOracleV2 & FxFrxETHOracleV2 & FxLSDOracleV2Base & FxSpotOracleBase & FxStETHOracleV2 & FxWBTCOracleV2

These contracts aggregate data from various price sources, further process it, and provide it for use by the f(x) protocol. The main function in this contract 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.

SpotPriceOracle

This contract retrieves spot price data from various price sources and converts it into the same precision value. The main function in this contract is as follows:

- `getSpotPrice()`

  Return the spot price sourced from various protocols such as Curve, Uniswap, and Balancer.

4. Audit Detail

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

4.1 Audit Process

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

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

4.2 Audit Result

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

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

4.3 Issues

4.3.1 Discussion on the risk of using spot prices from multiple sources.

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

Location

FxBTCDerivativeOracleBase.sol#L76

Description

The current oracle has modified the price scheme, replacing the old TWAP prices with spot prices from multiple sources, alleviating arbitrage behaviors caused by lagging TWAP prices during market volatility. When using the current multi-price-source spot price comparison scheme, the following issues need to be considered:

(1) Compared to the old scheme using TWAP prices, the spot prices from decentralized price sources in the current scheme are highly susceptible to manipulation. To address the risk of price manipulation, the current code has modified the corresponding price usage scheme: the lowest price from different price sources is used when minting fTokens or redeeming xTokens, while the highest price is used when minting xTokens or redeeming fTokens, ensuring that attackers cannot directly profit from manipulating prices. Although attackers cannot directly profit from price manipulation, they may cause potential losses to users' principals in some instances. Taking the example of a user minting fTokens, when an attacker manipulates the spot price of a certain price source using a sandwich attack method, resulting in a very low price (even close to 0), the user ultimately receives fTokens far below the actual quantity. The consequence of such an attack is that the principal lost by the user (especially a large holder) will be distributed among all holders of xTokens, who can redeem xTokens at that time to withdraw these funds. In this attack scenario, users minting fTokens bear the risk of losing their principal. In practice, users can mitigate such risks when calling mint/redeem fToken/xToken by setting a minimum amount for minting/redeeming funds.

(2) The current oracle utilizes spot prices from multiple price sources. While having more price sources can, to some extent, provide a more accurate reflection of real-time market prices, it also introduces a single point of failure risk from another perspective. In the current code logic, all decentralized price source data is used for comparison without any verification. If data from any price source becomes invalid (or manipulated), it will directly

impact the protocol's mint/redeem logic.

```
function getPrice()
  external
  view
  override
  returns (
    bool isValid,
    uint256 twap,
    uint256 minPrice,
    uint256 maxPrice
  )
{
  twap = _getBTCDerivativeUSDTwapPrice();
  (minPrice, maxPrice) = _getBTCDerivativeMinMaxPrice(twap);
  unchecked {
    isValid = (maxPrice - minPrice) * PRECISION < maxPriceDeviation *
minPrice;
  }
}

function _getBTCDerivativeMinMaxPrice(uint256 twap) internal view returns
(uint256 minPrice, uint256 maxPrice) {
  minPrice = maxPrice = twap;

  // @audit get spot price from other protocols
  uint256[] memory BTCDerivative_USD_prices =
getBTCDerivativeUSDSpotPrices();

  uint256 length = BTCDerivative_USD_prices.length;
  //@audit get max/min price
  for (uint256 i = 0; i < length; i++) {
    uint256 price = BTCDerivative_USD_prices[i];
    if (price > maxPrice) maxPrice = price;
    if (price < minPrice) minPrice = price;
  }
}
```

Status

The development team explained as follows:

Like the slippage protection measures taken during asset exchanges on most decentralized exchanges (DEX), the f(x) protocol also provides the same functionality during mint/redeem operations. When users interact with the f(x) protocol through the frontend web interface, the frontend assists users in setting the slippage. Suppose a user directly interacts with the contract and does not use slippage protection (setting it to 0). In that case, there is indeed a risk of sandwich attacks, which is considered a high-risk operation that the user fully understands. On the other hand, for pools with insufficient liquidity where malicious actors might manipulate prices downward, the f(x) protocol frontend provides warnings if the maximum and minimum differences exceed 1% (adjustable), giving users risk alerts.

As for the second point regarding single-point failures, the current option is to replace the corresponding data source.

4.3.2 Discussion on the return value of the function `_getLSDUSDTwap()`.

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

Location

FxEzETHOracleV2.sol#L54

Description

The function `_getLSDUSDTwap()` design intention is likely to fetch the price of the LSD-USD trading pair. Dividing the return value by the corresponding ezETH-ETH exchange rate may indicate a desire to convert the price of LSD-USD to the price of ezETH-USD. As suggested by the parameter names, `ezETH_ETH_RedStoneTwap * ETH_USD_ChainlinkTwap` already represents the price of ezETH-USD, so the subsequent division operation appears confusing.

```
function _getLSDUSDTwap() internal view virtual override returns
(uint256) {
    // @audit ezETH-ETH's price
    uint256 ezETH_ETH_RedStoneTwap =
ITwapOracle(RedStone_ezETH_ETH_Twap).getTwap(block.timestamp);
    // @audit ETH-USD's price
    uint256 ETH_USD_ChainlinkTwap = _getETHUSDTwap();
    // @audit The rate is ETH/ezETH's price
    (uint256 rate, , , ) =
IBalancerPool(Balancer_ezETH_Pool).getTokenRateCache(ezETH);
    unchecked {
      return (ezETH_ETH_RedStoneTwap * ETH_USD_ChainlinkTwap) / rate;
    }
  }
```

Status

The development team's explanation: The influence of the rate is still removed here to maintain consistency with the logic of the old version of the code.

4.3.3 Malicious attackers can disrupt normal redemption behavior by minting small amounts of xTokens to users.

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

Location

LeveragedTokenV2.sol#L137-L147

Description

The new f(x) protocol will restrict users' ability to mint xTokens to prevent atomic arbitrage in one transaction. Users cannot call the transfer function to transfer (redeem) tokens within half an hour (adjustable) after minting xTokens. It's worth noting that users can directly call the `mintXToken()` function to mint xTokens to a specified `_recipient` address. Restricting users from transferring tokens may lead to unexpected situations. Consider the following scenario: Suppose users cannot transfer xTokens within half an hour after minting them. Attackers could maliciously mint small amounts of xTokens to the user's address every half hour, preventing the user from calling the redeem function to redeem the base token. Particularly in cases of significant market price fluctuations, this malicious minting of xTokens for a specific `_recipient` directly impacts the user's ability to redeem tokens, potentially resulting in financial losses for the user.

```
function _beforeTokenTransfer(
  address from,
  address to,
  uint256
) internal virtual override {
  if (from == address(0)) {
    mintAt[to] = block.timestamp;
  } else if (block.timestamp - mintAt[from] < coolingOffPeriod) {
    revert ErrorTransferBeforeCoolingOffPeriod();
  }
}
```

Status

The development team has acknowledged this issue. Due to the significant impact on a critical functionality of f(x), the development team has decided to maintain the current code logic by not removing the delegated minting of xTokens feature. The team has also agreed to monitor on-chain transactions and, if attacks are detected, adjust the `coolingOffPeriod` parameter through governance mechanisms.

4.3.4 Discussion on handling the scale scheme.

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

Location

SpotPriceOracle.sol#L146-L151

SpotPriceOracle.sol#L164-L165

SpotPriceOracle.sol#L189-L190

Description

It's advisable to call the `sync()` function of the Uniswap V2 contract before invoking the `getReserves()` function to ensure consistency between token balance records and the actual quantities the contract holds.

Additionally, due to potential discrepancies in decimals among different tokens, it's crucial to unify them to the same scale to avoid rounding errors when handling prices. Typically, scaling issues are addressed by reading the decimals of each token in the pool. The current approach requires manual input of parameters by the administrator, increasing the risk of errors. Generally, it's safer to handle scaling issues by reading the decimals of each token in the pool.

```
function _getSpotPriceByUniswapV2(uint256 encoding) internal view returns
(uint256) {
    address pool = _getPool(encoding);
    uint256 base_index = (encoding >> 160) & 1;

    // @audit  it is advisable to first call the sync() function to
update the token balance records
    (uint256 r_base, uint256 r_quote, ) =
IUniswapV2Pair(pool).getReserves();
    if (base_index == 1) {
      (r_base, r_quote) = (r_quote, r_base);
    }
    // @audit it appears to address scaling issues, but since the
parameters are provided by the administrator, their accuracy cannot be
guaranteed
    r_base *= 10**((encoding >> 161) & 255);
    r_quote *= 10**((encoding >> 169) & 255);

    return (r_quote * PRECISION) / r_base;
  }
```

Status

The development team explained: Since the tokens currently used in the protocol are not rebase tokens, there is no need to call the `sync()` function to update the relevant parameters. Furthermore, considering the fact that it is impossible to prevent the malicious manipulation of spot prices (manipulating spot prices will not affect the normal operation of the f(x) protocol), calling the `sync()` function would not serve much purpose and would instead waste more gas.

The purpose of manually passing the parameters by the administrator is to reduce gas consumption.

4.3.5 Discussion on the `balances` parameter passed to the function `calculateInvariant()`.

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

Location

SpotPriceOracle.sol#L239

Description

The function `_calculateInvariant()` is refactored based on the logic of the _get_D() function in the Curve protocol. Here, the parameter `balances` in the function `_calculateInvariant()` corresponds to the `_xp` parameter in the _get_D() function, which should represent the balances adjusted for rates. However, it seems that the influence of rates was not considered when calling the function.

```
function _getSpotPriceByCurvePlain(uint256 encoding) internal view
returns (uint256) {
    address pool = _getPool(encoding);
    uint256 tokens = ((encoding >> 160) & 7) + 1;
    uint256 base_index = (encoding >> 163) & 7;
    uint256 quote_index = (encoding >> 166) & 7;
    uint256 has_amm_precise = (encoding >> 169) & 1;
    uint256 amp;
    // curve's precision is 100, and we use 1000 in this contract.
    if (has_amm_precise == 1) {
      amp = ICurvePlainPool(pool).A_precise() * 10;
    } else {
      amp = ICurvePlainPool(pool).A() * 1000;
    }
    encoding >>= 170;
    uint256[] memory balances = new uint256[](tokens);
    for (uint256 i = 0; i < tokens; ++i) {
      balances[i] = ICurvePlainPool(pool).balances(i);
      // scale to 18 decimals
      balances[i] *= 10**(encoding & 255);
      encoding >>= 8;
    }
    // @audit corresponding function in the Curve protocol would be
`_get_D()`
    uint256 invariant = StableMath.calculateInvariant(amp, balances);
    return StableMath.calculateSpotPrice(base_index, quote_index, amp,
invariant, balances);
  }
```

Status

The development team explains that the influence of rates was indeed considered when handling scaling. They also confirm that when administrators input scale parameters, they will ensure the correct configuration of relevant parameters.

5. Conclusion

After auditing and analyzing the Twap And Spot Price Oracle of f(x) Protocol, 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.

SECBIT Lab is devoted to constructing a common-consensus, reliable, and ordered blockchain economic entity.

https://secbit.io

audit@secbit.io

@secbit_io