Security Audit Report

f(x) Protocol New Features by AladdinDAO

SECBIT

November 29, 2023

1. Introduction

The AladdinDAO is a decentralized network to shift crypto investments from venture capitalists to the wisdom of crowds through collective value discovery. As a part of the AladdinDAO ecosystem, the f(x) protocol creates two new ETH derivative assets, one with stablecoin-like low volatility called fractional ETH (fETH) and the second a leveraged long ETH perpetual token called leveraged ETH (xETH). The audit includes the following four main components:

1. **Gauge Module Development:** Drawing inspiration from the Curve protocol, the f(x) protocol has developed a new gauge module. Users can deposit their LP tokens into the corresponding gauge to earn FXN token rewards. Additionally, similar to the ve model in Curve, f(x) supports users holding veFXN tokens, enhancing their earnings through a boosting mechanism.
2. **Incentive Mechanism Modification:** The f(x) protocol has modified the incentive mechanisms for redeeming fETH and xETH tokens.
3. **Earnings Conversion Logic:** Introducing a new logic for converting earnings, veFXN tokens' rewards are converted from stETH to wstETH before distribution.
4. **Support for External Protocols:** Users can lock their FXN tokens and store them in external protocols like Convex or StakeDAO to generate additional earnings.
5. **Withdraw principal at any time:** After the administrator adjusts the contract parameters, users can withdraw their principal deposited in the rebalance pool contract at any time, without waiting for the lock-up period to expire.

SECBIT Labs conducted an audit from September 18 to November 29, 2023, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that this f(x) contract update has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising (see part 4 for details).

Type	Description	Level	Status
Design & Implementation	4.3.1 The user's benefit data cannot be updated, which prevents them from claiming rewards.	Medium	Fixed
Design & Implementation	4.3.2 When transferring LP tokens between users, the profit calculation for both parties can be incorrect.	Medium	Fixed
Design & Implementation	4.3.3 Discussing the implementation logic of the `balanceOf()` function.	Info	Discussed
Design & Implementation	4.3.4 The deducted amount in the `boostFrom()` function does not match the actual amount used.	Info	Discussed
Design & Implementation	4.3.5 The code incorrectly records the actual number of voting weights revoked by the user.	Info	Discussed
Gas optimization	4.3.6 Optimize the code logic to save gas for most users.	Info	Discussed
Design & Implementation	4.3.7 Discuss the distribution of additional rewards users obtain when minting xETH tokens through the `wrap()` function.	Info	Discussed
Design & Implementation	4.3.8 Malicious users can forge the `context` parameter to bypass restriction conditions.	Info	Discussed
Design & Implementation	4.3.9 Attackers can take away rewards belonging to bulk deposit users at very low costs.	Low	Fixed
Design & Implementation	4.3.10 The `manage()` function might fail to execute due to insufficient reward funds in the contract.	Medium	Fixed
Design & Implementation	4.3.11 The `_operator` has not been granted authorization to transfer the `_rewardToken` tokens from the current contract.	Medium	Fixed
Gas optimization	4.3.12 Add value validation for the parameter `_unvested` to save gas.	Info	Fixed
Design & Implementation	4.3.13 Users cannot directly withdraw cvxFXN tokens or sdFXN tokens.	Info	Discussed
Design & Implementation	4.3.14 Users may potentially lose incentives within the sdFXN gauge.	Info	Discussed
Design & Implementation	4.3.15 Calling the `emaValue()` function may fail under specific conditions, consequently rendering critical functions of the f(x) protocol, such as mint, redeem, and liquidate, unusable.	Medium	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

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

- Smart contract code
  - initial review commit
    - shared ve proxy and liquidity gauge
    - two new features for f(x)
    - ConvexCurveManager
    - manageable vesting
    - rebalance pool with boost
  - final review commit
    - shared ve proxy and liquidity gauge
    - manageable vesting
    - rebalance pool with boost

Notice: This audit will not cover the newly added contracts in rebalance pool with boost. We will finish the audit for those contracts in coming weeks.

2.2 Contract List

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

Name	Lines	Description
MultipleRewardAccumulator.sol	140	Abstract contract for batch reward processing.
LinearMultipleRewardDistributor.sol	101	Abstract contract for batch reward distribution.
LinearReward.sol	49	Contract for recording income information for each period.

LinearRewardDistributor.sol	51	Abstract contract for individual reward distribution processing.
VotingEscrowBoost.sol	262	Holders of veFXN tokens can use the boost mechanism in this contract to increase their earnings.
VotingEscrowProxy.sol	31	The administrator uses this contract to adjust the balance of veTokens considering delegating.
LiquidityGauge.sol	311	A contract where users deposit staking tokens to earn profits, and users holding veTokens can earn higher yields.
SharedLiquidityGauge.sol	125	A contract allowing delegated voting.
ConvexCurveManager.sol	144	A contract for managing user deposits of assets.
LiquidityManagerBase.sol	92	A contract for managing user deposits of assets.
WordCodec.sol	49	Library for encoding and decoding values stored inside a 256-bit word. Typically used to pack multiple values in a single storage slot, saving gas by performing fewer storage accesses.
ConvexCurveManagerImmutable.sol	134	A contract for managing user assets and earnings.
LiquidityManagerBaseImmutable.sol	89	A contract for managing user assets and earnings.
Market.sol	491	A contract that implements the core logic of the f(x) protocol.
Treasury.sol	400	A contract to store the baseToken, where the core functions can only be called by the Market contract.
RebalancePool.sol	548	A Rebalance Pool to liquidate collateralized debt positions that fall below the minimum collateralization ratio.
RebalancePoolRegistry.sol	35	A contract for managing the registration of pools.
RebalancePoolSplitter.sol	103	A contract that distributes rewards based on a specified splitter.
RebalanceWithBonusToken.sol	27	To provide a contract liquidation interface for any user and offer liquidation rewards to the caller.
StETHAndxETHWrapper.sol	40	Peripheral contract of the f(x) protocol, allowing users to mint and redeem xETH tokens.
FxGateway.sol	181	Extend the use cases of the fx protocol to allow users to participate in the protocol using approved tokens.
GeneralTokenConverter.sol	636	A utility contract for handling token swaps.
LidoConverter.sol	77	A utility contract to exchange stETH and wstETH tokens.
CvxFxnVestingManager.sol	40	Unvested FXN tokens can be deposited through this contract to earn yield under the Convex protocol.
PlainVestingManager.sol	25	One of the FXN token handling strategies. Users selecting this strategy will directly receive unlocked FXN tokens.

SdFxnVestingManager.sol	43	Unvested FXN tokens can be deposited through this contract to earn yield under the StakeDao protocol.
ManageableVesting.sol	180	A contract for handling users' FXN tokens that are pending vesting.
VestingManagerProxy.sol	19	Proxy contract managing user funds and earnings.

Notice: This audit specifically focuses on the new code introduced in above contracts.

3. Contract Analysis

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

3.1 Role Classification

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

- Governance Account
  - Description

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

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

    Users participate in the f(x) Protocol.
  - Authority
    - Mint fETH / xETH with authorized tokens
    - Redeem base tokens with fETH / xETH tokens

        - Add base tokens to increase the collateralization ratio
        - Liquidate fETH tokens to increase the collateralization ratio
        - Swap between fETH and xETH
        - Deposit LP token
    - 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 new version of the f(x) protocol has introduced several new features. Users can deposit LP tokens, such as fxnETH-ETH LP tokens, fETH-crvUSD LP tokens, etc., to earn FXN token rewards. Additionally, users holding veFXN tokens will receive higher yields. 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 three parts:

LiquidityGauge

This contract allows users to deposit specified staking tokens, and in return, they receive FXN token rewards. Like ve model in the Curve protocol, users holding veFXN tokens can also earn higher yields at a proportional rate.

The main functions in this contract are as follows:

- `deposit()`

  Users can deposit some staking tokens to this contract.
- `withdraw()`

  Users can withdraw some staking tokens from this contract.
- `user_checkpoint()`

  Update the snapshot for some user.

VotingEscrowBoost

Users holding veFXN tokens can authorize their shares to other users, aiming to achieve higher yields. The main functions in this contract are as follows:

- `boost()`

  Users holding veFXN tokens can delegate their power to other users.

- `unboost()`

  Users holding veFXN tokens can cancel their power to other users.

ManageableVesting, CvxFxnVestingManager, PlainVestingManager and SdFxnVestingManager

Users with locked FXN tokens have the freedom to choose from three different strategies: They can deposit their pending unlocked FXN tokens into the Convex protocol to earn rewards from the protocol. They can deposit their pending unlocked FXN tokens into the StakeDao protocol to earn rewards from that protocol. After the linear release of FXN tokens, users can choose to withdraw them directly.

The main functions in these contracts are as follows:

- `claim()`

  When users withdraw currently unlocked tokens, the type of funds they receive varies based on the strategy they have chosen.

- `manage()`

  Users can deposit unclaimed FXN tokens into the Convex or StakeDao protocols using 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 user's benefit data cannot be updated, which prevents them from claiming rewards.

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

Location

LiquidityGauge.sol

Description

The contract uses the `userSnapshot` parameter to record user profit information. Within the `_checkpoint()` function, user profits are updated using a `memory` type variable `_cachedUserSnapshot`, which avoids repeatedly reading data from the blockchain, thus reducing user costs. However, after completing the update of user profit data, it is not written to `userSnapshot[_account`. It means that the updated data will not be stored on the blockchain, making users unable to claim rewards.

```
function _checkpoint(address _account) internal virtual override {
    // checkpoint extra rewards
    MultipleRewardAccumulator._checkpoint(_account);

    RewardSnapshot memory _cachedSnapshot = snapshot;
    InflationParams memory _prevInflationParams = inflationParams;
    InflationParams memory _nextInflationParams = _prevInflationParams;

    ......

    // update integral for user snapshot
    if (_account != address(0)) {
      uint256 _workingBalance = workingBalanceOf[_account];

      // @audit using `memory` means that the updated parameters below
will not be stored on the blockchain.
      UserRewardSnapshot memory _cachedUserSnapshot =
userSnapshot[_account];
      _cachedUserSnapshot.rewards.pending += uint128(
        (_workingBalance * uint256(_cachedSnapshot.integral -
_cachedUserSnapshot.checkpoint.integral)) /
          REWARD_PRECISION
      );
      _cachedUserSnapshot.checkpoint.integral = _cachedSnapshot.integral;
      _cachedUserSnapshot.checkpoint.timestamp = uint64(block.timestamp);
    }
  }
```

Suggestion

After updating user data, write this data to the blockchain.

```
  function _checkpoint(address _account) internal virtual override {
    // checkpoint extra rewards
    MultipleRewardAccumulator._checkpoint(_account);

    RewardSnapshot memory _cachedSnapshot = snapshot;
    InflationParams memory _prevInflationParams = inflationParams;
    InflationParams memory _nextInflationParams = _prevInflationParams;

    ......

    // update integral for user snapshot
    if (_account != address(0)) {
      uint256 _workingBalance = workingBalanceOf[_account];

      UserRewardSnapshot memory _cachedUserSnapshot =
userSnapshot[_account];
      _cachedUserSnapshot.rewards.pending += uint128(
        (_workingBalance * uint256(_cachedSnapshot.integral -
_cachedUserSnapshot.checkpoint.integral)) /
          REWARD_PRECISION
      );
      _cachedUserSnapshot.checkpoint.integral = _cachedSnapshot.integral;
      _cachedUserSnapshot.checkpoint.timestamp = uint64(block.timestamp);
```

```
        //@audit add the following code
        userSnapshot[_account] = _cachedUserSnapshot;
      }
    }
```

Status

The development team has adopted the suggestion and fixed this issue in commit *8cc9d21*.

4.3.2 When transferring LP tokens between users, the profit calculation for both parties can be incorrect.

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

Location

SharedLiquidityGauge.sol

Description

When users deposit staking tokens into the contract, they receive an equivalent amount of LP tokens as proof of their shares. Since these LP tokens are standard ERC20 tokens, they can be transferred to different addresses. Let's assume there are two users, Alice and Bob. If Alice transfers her 100 LP tokens to Bob using the `ERC20.transfer()` function, the specific calculation logic can be divided into the following four steps:

```
// @audit https://github.com/OpenZeppelin/openzeppelin-contracts-
upgradeable/blob/v4.9.0/contracts/token/ERC20/ERC20Upgradeable.sol#L118-
L122

function transfer(address to, uint256 amount) public virtual override
returns (bool) {
        address owner = _msgSender();
        _transfer(owner, to, amount);
        return true;
}

function _transfer(address from, address to, uint256 amount) internal
virtual {
        require(from != address(0), "ERC20: transfer from the zero
address");
        require(to != address(0), "ERC20: transfer to the zero address");
        //@audit call this function first
        _beforeTokenTransfer(from, to, amount);

        uint256 fromBalance = _balances[from];
        require(fromBalance >= amount, "ERC20: transfer amount exceeds
balance");
        unchecked {
            _balances[from] = fromBalance - amount;
            // Overflow not possible: the sum of all balances is capped
by totalSupply, and the sum is preserved by
            // decrementing then incrementing.
            _balances[to] += amount;
        }

        emit Transfer(from, to, amount);

        _afterTokenTransfer(from, to, amount);
    }
```

Call the `_beforeTokenTransfer()` function. Since both sender and recipient addresses are not 0, the code will first update the staked balances of Alice and Bob, specifically updating the `sharedBalanceOf[Alice]` and `sharedBalanceOf[Bob]`.

```
function _beforeTokenTransfer(
    address _from,
    address _to,
    uint256 _amount
  ) internal virtual override nonReentrant {
    if (_from != address(0) && _amount > 0) {
      address _ownerFrom = _stakerVoteOwner[_from];
      if (_ownerFrom != address(0)) {
        unchecked {
          sharedBalanceOf[_ownerFrom] -= _amount;
        }
      }
    }

    if (_to != address(0) && _amount > 0) {
      address _ownerTo = _stakerVoteOwner[_to];
      if (_ownerTo != address(0)) {
        unchecked {
          sharedBalanceOf[_ownerTo] += _amount;
        }
      }
    }

    // no need to checkpoint on mint or burn or transfer to self
    if (_from == address(0) || _to == address(0) || _from == _to ||
_amount == 0) return;

    _checkpoint(_from);
    _checkpoint(_to);

    _updateWorkingBalance(_from);
    _updateWorkingBalance(_to);
  }
```

Call the `_checkpoint()` function to update the earnings of Alice and Bob. It's important to note that the `_checkpoint()` function calculates user earnings by directly reading the value of `workingBalanceOf[_account]`, which will be updated in the subsequent `_updateWorkingBalance()` function.

```
// @audit https://github.com/AladdinDAO/aladdin-v3-
contracts/blob/19036c7a3123e02052cf685a80b93781c6dfe4f4/contracts/voting-
escrow/gauges/liquidity/LiquidityGauge.sol#L258
function _checkpoint(address _account) internal virtual override {
    ......

      _cachedSnapshot.timestamp = uint64(block.timestamp);
      snapshot = _cachedSnapshot;
    }

    // update integral for user snapshot
    if (_account != address(0)) {
      // @audit get working balance directly
      uint256 _workingBalance = workingBalanceOf[_account];
      UserRewardSnapshot memory _cachedUserSnapshot =
userSnapshot[_account];
      _cachedUserSnapshot.rewards.pending += uint128(
        (_workingBalance * uint256(_cachedSnapshot.integral -
_cachedUserSnapshot.checkpoint.integral)) /
          REWARD_PRECISION
      );
      _cachedUserSnapshot.checkpoint.integral = _cachedSnapshot.integral;
      _cachedUserSnapshot.checkpoint.timestamp = uint64(block.timestamp);
    }
  }
```

Call the _updateWorkingBalance() function to update the user's working balance. It's worth noting that the working balance is related to the number of veTokens and LP tokens. At this point, because Alice's 100 LP tokens have not yet been transferred to Bob's account (the _updateWorkingBalance() function is still within the _beforeTokenTransfer() function and has not updated the balances of Alice and Bob), these 100 LP tokens are still considered part of Alice's working balance.

```
// https://github.com/AladdinDAO/aladdin-v3-
contracts/blob/19036c7a3123e02052cf685a80b93781c6dfe4f4/contracts/voting-
escrow/gauges/liquidity/LiquidityGauge.sol#L470
function _updateWorkingBalance(address _account) internal {
    uint256 _workingBalance = _computeWorkingBalance(_account);
    uint256 _oldWorkingBalance = workingBalanceOf[_account];
    uint256 _workingSupply = workingSupply + _workingBalance -
_oldWorkingBalance;
    workingBalanceOf[_account] = _workingBalance;
    workingSupply = _workingSupply;

    emit UpdateLiquidityLimit(_account, balanceOf(_account),
totalSupply(), _workingBalance, _workingSupply);
  }

function _computeWorkingBalance(address _account) internal view virtual
returns (uint256) {

    // @audit get veToken balance
    uint256 _veBalance = _getUserVeBalance(_account);

    uint256 _veSupply = IVotingEscrow(ve).totalSupply();

    // @audit get lp token balance
    uint256 _balance = balanceOf(_account);
    uint256 _supply = totalSupply();

    uint256 _workingBalance = (_balance * TOKENLESS_PRODUCTION) / 100;
    if (_veSupply > 0) {
      _workingBalance += (((_supply * _veBalance) / _veSupply) * (100 -
TOKENLESS_PRODUCTION)) / 100;
    }
    if (_workingBalance > _balance) {
      _workingBalance = _balance;
    }

    return _workingBalance;
    }
```

Updating the LP token ledger balances for both Alice and Bob occurs after the execution of the `_beforeTokenTransfer()` function. Therefore, this transfer of 100 LP tokens from Alice to Bob will not immediately impact the changes in their working balances. It means that Alice will continue to earn rewards from these 100 LP tokens, even though she has transferred them to Bob.

In summary, the issue arises from the transfer function not immediately modifying the user's LP token ledger balance when calling `_beforeTokenTransfer()`. This mismatch leads to a lack of synchronization between user earnings and account balance changes. In a typical scenario, the user's LP token ledger balance should be updated first, followed by the update of the working balance to ensure proper synchronization.

Status

The development team added the `_afterTokenTransfer()` function in commit *cc4ffd6*, adjusting the code logic sequence and fixing this issue.

4.3.3 Discussing the implementation logic of the `balanceOf()` function.

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

Location

VotingEscrowBoost.sol

Description

The return value of the function `_balanceOf()` represents the voting weight currently held by the account, including the portion delegated on its behalf. From the comments in the `balanceOf()` function and `adjustedVeBalance()` function, it is apparent that the latter aligns better with the intended semantics. According to the function comments, the return value of the `balanceOf()` function should reflect the current quantity of tokens the user holds. It should not include the portion delegated on their behalf. Therefore, ensuring that the implementation of the `balanceOf()` function aligns with the design requirements is crucial.

```
/// @notice Returns the amount of tokens owned by `account`.
function balanceOf(address account) external view returns (uint256);
```

```
/// @notice Return the ve balance considering delegating.
/// @param account The address of user to query.
function adjustedVeBalance(address account) external view returns
(uint256);


function balanceOf(address _account) external view returns (uint256) {
    return _balanceOf(_account);
  }

/// @inheritdoc IVotingEscrowBoost
function adjustedVeBalance(address _account) external view returns
(uint256) {
    return _balanceOf(_account);
  }


function _balanceOf(address _account) private view returns (uint256) {
    uint256 _amount = IVotingEscrow(ve).balanceOf(_account);

    (Point memory p, ) = _checkpoint(_account, true);
    _amount -= uint256(p.bias - p.slope * (block.timestamp - p.ts));

    (p, ) = _checkpoint(_account, false);
    _amount += uint256(p.bias - p.slope * (block.timestamp - p.ts));

    return _amount;
  }
```

Status

The development team has confirmed that this design choice was made to align with the logic of the Curve protocol.

4.3.4 The deducted amount in the `boostFrom()` function does not match the actual amount used.

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

Location

VotingEscrowBoost.sol

Description

Authorized third-party users can use the boostFrom() function to assist veToken holders in boosting their voting power. This function first calls the _spendAllowance() function to deduct the authorized allowance of _amount from the _owner user. It then proceeds to call the _boost() function to carry out the delegation operation. Due to rounding errors in Solidity arithmetic operations, the actual amount of funds bias that the _owner user delegates may be slightly less than the _amount, resulting in an extra deduction from the _owner's allowance.

Furthermore, one should note that the input parameter amount for both the boost() and boostFrom() functions may not represent the actual quantity of ve tokens used for boosting, as the actual utilized quantity bias might be slightly smaller than amount.

```
/// @notice Boost ve balance to other user.
/// @param receiver The address of recipient.
/// @param amount The amount of ve balance to boost.
/// @param endtime The end timestamp of this boost.
function boost(
  address receiver,
  uint256 amount,
  uint256 endtime
) external;

/// @notice Boost ve balance to other user on behalf of another user.
/// @param owner The address of ve balance owner.
/// @param receiver The address of recipient.
/// @param amount The amount of ve balance to boost.
/// @param endtime The end timestamp of this boost.
function boostFrom(
  address owner,
  address receiver,
  uint256 amount,
  uint256 endtime
) external;


function boostFrom(
    address _owner,
    address _receiver,
    uint256 _amount,
    uint256 _endtime
  ) external {
    address spender = _msgSender();

    //@audit actual used amount might be slightly less than `_amount`.
    _spendAllowance(_owner, spender, _amount);
    _boost(_owner, _receiver, _amount, _endtime);
  }

function _boost(
    address _owner,
    address _receiver,
    uint256 _amount,
    uint256 _endtime
  ) private {
    ......

    // calculate slope and bias being added
    uint112 slope = uint112(_amount / (_endtime - block.timestamp));

    // @audit due to rounding error, the bias might be smaller than
    // '_amount'
    uint112 bias = uint112(slope * (_endtime - block.timestamp));

    // update delegated point
    p.bias += bias;
    p.slope += slope;

    ......
  }
```

Status

This issue has been discussed.

4.3.5 The code incorrectly records the actual number of voting weights revoked by the user.

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

Location

VotingEscrowBoost.sol

Description

Users can revoke their delegated voting power through the `unboost()` function, with the input parameter `_amount` indicating the quantity of voting power the user wants to revoke. Based on the specific code logic, the actual voting power revoked by the user is time-dependent. The user's revocable voting power linearly decreases over time. The parameter `_item.cancelAmount` should represent the actual quantity of voting power revoked by the user. However, the current code directly adds the quantity the user wants to cancel, which does not align with the actual scenario.

```
/// @notice Cancel an existing boost.
/// @param index The index of in the boost lists.
/// @param amount The amount of boost to cancel.
function unboost(uint256 index, uint128 amount) external;

function unboost(uint256 _index, uint128 _amount) external {
    address _owner = _msgSender();
    if (_index >= boostLength(_owner)) revert IndexOutOfBound();

    BoostItem memory _item = boosts[_owner][_index];

    //@audit the actual withdrawal value might be smaller than the
specified value
    _item.cancelAmount += _amount;
    if (_item.cancelAmount > _item.initialAmount) revert
CancelBoostExceedBalance();
    if (_item.endTime <= block.timestamp) revert CancelExpiredBoost();

    // update amount based on current timestamp
    // @audit the actual quantity used for withdrawal.
    _amount -= uint128((uint256(_amount) * (block.timestamp -
_item.startTime)) / (_item.endTime - _item.startTime));

    // checkpoint delegated point
    Point memory p = _checkpointWrite(_owner, true);

   ......
  }
```

Status

Based on the explanation from the development team, the parameter `cancelAmount` records the canceled quantity corresponding to the initial authorization amount that the user provided. Therefore, the input `amount` to be canceled here is also based on the user's initial authorized quantity.

4.3.6 Optimize the code logic to save gas for most users.

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

Location

VotingEscrowBoost.sol

Description

In most cases, the `p.ts == block.timestamp` condition is met after the calculation in the `_checkpointWrite()` function, and the obtained `p.bias` represents the user's currently utilized voting power. Therefore, for most users, the expression inside the `if` condition can be simplified to save gas for the caller.

However, in extreme cases where a user has not performed a boost/unboost operation for 255 weeks (approximately 4.89 years), meaning `p.ts > block.timestamp`, the current code logic should be retained.

```
function _boost(
    address _owner,
    address _receiver,
    uint256 _amount,
    uint256 _endtime
  ) private {
    if (_amount == 0) revert BoostZeroAmount();
    if (_endtime <= block.timestamp) revert
EndTimeSmallerThanCurrentTimestamp();
    if (_endtime % WEEK != 0) revert EndTimeNotAlignedWithWeek();
    if (_endtime > IVotingEscrow(ve).locked__end(_owner)) revert
EndTimeExceedLockEnd();

    // checkpoint delegated point
    Point memory p = _checkpointWrite(_owner, true);
    //@audit in the vast majority of cases, `p.ts == block.timestamp`.
    if (_amount > IVotingEscrow(ve).balanceOf(_owner) - (p.bias - p.slope
* (block.timestamp - p.ts))) {
      revert BoostExceedBalance();
    }

    // calculate slope and bias being added
    uint112 slope = uint112(_amount / (_endtime - block.timestamp));
    uint112 bias = uint112(slope * (_endtime - block.timestamp));

   ......
  }


function _checkpointWrite(address _account, bool _isDelegated) internal
returns (Point memory p) {
    uint256 dbias;
    (p, dbias) = _checkpoint(_account, _isDelegated);

    // received boost
    if (!_isDelegated && dbias > 0) {
      emit Transfer(_account, address(0), dbias);
    }
  }

function _checkpoint(address _account, bool _isDelegated) internal view
returns (Point memory p, uint256 dbias) {
    p = _isDelegated ? delegated[_account] : received[_account];
    if (p.ts == 0) {
      p.ts = uint32(block.timestamp);
    }
    if (p.ts == block.timestamp) {
      return (p, dbias);
    }

    uint256 ts = (p.ts / WEEK) * WEEK;

    // @audit the loop might execute 256 times only in extreme scenarios.
    for (uint256 i = 0; i < 255; i++) {
      ts += WEEK;
      uint256 _slopeChange = 0;
      if (ts > block.timestamp) {
        ts = block.timestamp;
      } else {
        _slopeChange = _isDelegated ? delegatedSlopeChanges[_account][ts]
: receivedSlopeChanges[_account][ts];
      }

      uint112 _amount = p.slope * uint112(ts - p.ts);
      dbias += uint256(_amount);
      p.bias -= _amount;
      p.slope -= uint112(_slopeChange);
      p.ts = uint32(ts);

      if (p.ts == block.timestamp) {
        break;
      }
    }
  }
```

Status

The development team has confirmed the issue. The current code logic does not impact the security of the code and handles extreme scenarios. Therefore, the developer team has decided to retain the current design.

4.3.7 Discuss the distribution of additional rewards users obtain when minting xETH tokens through the `wrap()` function.

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

Location

StETHAndxETHWrapper.sol

Description

When minting xETH tokens in this contract, there might be additional rewards that can only be sent to the specified `platform` address. If the stETH used to mint xETH tokens is provided by a user, the additional rewards should also go to the user rather than the specified `platform` address. It's crucial to confirm whether the extra awards are exclusively designated to be transferred to the `platform` address.

```
function wrap(uint256 _amount) external override returns (uint256) {
    uint256 _bonus;
    (_amount, _bonus) = IMarket(market).mintXToken(_amount,
address(this), 0);
    IERC20(dst).safeTransfer(msg.sender, _amount);

    // transfer bonus to platform
    if (_bonus > 0) {
      IERC20(src).safeTransfer(platform, _bonus);
    }
    return _amount;
  }
```

Status

According to the explanation from the developer team, This contract is private, and users are not expected to invoke it. As designed, when called through the `rebalance pool` function, the bonus will be set to 0, meaning no rewards will be distributed in practice.

4.3.8 Malicious users can forge the `context` parameter to bypass restriction conditions.

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

Location

GeneralTokenConverter.sol

Description

When users utilize the swap function of Uniswap V3 within the contract, they need to use the `fallback()` function to transfer their funds to the Uniswap V3 contract under this contract's control. It's worth noting that the `context` parameter in the `fallback()` function is provided by the user. Malicious users could potentially bypass the `address(_context) == _msgSender()` condition by crafting a phony `context` parameter.

```
fallback() external payable {
    uint256 _context = context;
    if (address(_context) == _msgSender() || _context == 1) {
      // handle uniswap v3 swap callback or uniswap v3 quote callback
      // | 4 bytes |   32 bytes    |    32 bytes    |    32 bytes   |    32
bytes   | 32 bytes |
      // |    sig    | amount0Delta | amount1Delta | data.offset |
data.length |  tokenIn |
      int256 amount0Delta;
      int256 amount1Delta;
      address tokenIn;
      assembly {
        amount0Delta := calldataload(4)
        amount1Delta := calldataload(36)
        tokenIn := calldataload(132)
      }
      (uint256 amountToPay, uint256 amountReceived) = amount0Delta > 0
        ? (uint256(amount0Delta), uint256(-amount1Delta))
        : (uint256(amount1Delta), uint256(-amount0Delta));
      if (_context == 1) {
        assembly {
          let ptr := mload(0x40)
          mstore(ptr, amountReceived)
          revert(ptr, 32)
        }
      } else {
        IERC20(tokenIn).safeTransfer(address(_context), amountToPay);
      }
    } else {
      revert("invalid call");
    }
  }
```

Status

The developer team has confirmed that this contract, serving as an auxiliary contract for exchanging tokens, does not hold any assets itself. Therefore, attackers cannot profit from it.

4.3.9 Attackers can take away rewards belonging to bulk deposit users at very low costs.

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

Location

ConvexCurveManagerImmutable.sol

ConvexCurveManager.sol

Description

According to the code design requirements, users could decide whether to deposit LP tokens into the Convex protocol. If so, they can get a bonus from the profits under the `harvest` function. Consider the following scenario:

A user first calls the `harvest()` function to retrieve protocol earnings. At this point, all funds in this contract will be deposited into the Convex protocol, and a portion of the earnings retrieved will be retained as a bounty reward for batch depositing users. Next, the user directly deposits a small amount of LP tokens into the contract and then calls the `manage()` function. It will withdraw all funds reserved for bounties in the contract.

Based on the above approach, as long as a user calls the `harvest()` function, arbitrageurs can deposit a small amount of LP tokens directly into the contract and then call the `manage()` function to withdraw these bounty rewards. Such actions could disrupt the normal logic, and users who deposit in batches into the Convex protocol would lose their incentives.

```
function manage(address _receiver) public override {

    uint256 _balance = IERC20(token).balanceOf(address(this));
    // @audit if a user directly transfers a small amount of tokens to
the contract, they can bypass this condition.
    if (_balance == 0) return;

    // deposit to booster
    IConvexBooster(BOOSTER).deposit(pid, _balance, true);

    // send incentive
    uint256 _length = rewards.length;
    for (uint256 i = 0; i < _length; ++i) {
      address _rewardToken = rewards[i];
      // @audit get the incentive
      uint256 _incentive = incentive[_rewardToken];
      if (_incentive > 0) {
        IERC20(_rewardToken).safeTransfer(_receiver, _incentive);
        incentive[_rewardToken] = 0;
      }
    }
    }
```

Status

The development team has modified the code logic to no longer provide incentives when users call the `manage()` function in commit *8cc9d21*.

4.3.10 The `manage()` function might fail to execute due to insufficient reward funds in the contract.

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

Location

ConvexCurveManagerImmutable.sol

ConvexCurveManager.sol

Description

`incentive[_rewardToken]` parameter records the bounty amount that users can receive when depositing in batches under the Convex protocol. There is an error in the calculation logic for this parameter.

To simplify the calculation, let's assume the following parameter values in the contract:

$$\_managerRatio = 10^8(10\%),\ \ \_harvesterRatio = 0,\ \ \_rewardToken = address(cvxcrv)$$

Consider the following scenario:

The user calls the `harvest()` function to claim 100 CVXCRV tokens as profit. At this point, there are 10 CVXCRV tokens left in the contract as a reward for calling the `manage()` function.

$$\begin{gathered} \_rewardAmount = 100 \\ \_incentive = 100 \times 10\% = 10 \\ incentive[cvxcrv] = 10 \end{gathered} \tag{1}$$

Assuming no user has called the `manage()` function. At this point, when the user calls the `harvest()` function again, they also receive 100 CVXCRV tokens as profit. Note that at this point, 110 - 11 = 99 CVXCRV tokens will be transferred to the operator contract. Currently, there are only 11 CVXCRV tokens left in the contract.

$$\begin{aligned} \_rewardAmount &= 100 + 10 = 110 \\ \_incentive &= 110 \times 10\% = 11 \\ incentive[cvxcrv] &= 10 + 11 = 21 \end{aligned} \tag{2}$$

If a user calls the `manage()` function at this point, intending to transfer all LP tokens from the contract to the Convex contract, they will receive a bounty incentive of `incentive[cvxcrv] = 21` CVXCRV tokens. However, there are currently only 11 CVXCRV tokens left in the contract. The `IERC20(_rewardToken).safeTransfer()` function will fail due to insufficient funds.

```
function harvest(address _receiver) external {
    // try to deposit first
    uint256 _balance = IERC20(token).balanceOf(address(this));
    if (_balance > 0) {
      IConvexBooster(BOOSTER).deposit(pid, _balance, true);
    }

    // harvest
    IConvexBasicRewards(rewarder).getReward();

    // distribute rewards
    uint256 _harvesterRatio = getHarvesterRatio();
    uint256 _managerRatio = getManagerRatio();
    uint256 _length = rewards.length;
    for (uint256 i = 0; i < _length; ++i) {
      address _rewardToken = rewards[i];

      //@audit read the balance in this contract
      uint256 _rewardAmount =
IERC20(_rewardToken).balanceOf(address(this));
      if (_rewardAmount == 0) continue;

      unchecked {
        uint256 _incentive = (_rewardAmount * _managerRatio) /
FEE_PRECISION;
        // @audit accumulate rewards
        if (_incentive > 0) incentive[_rewardToken] += _incentive;

        uint256 _bounty = (_rewardAmount * _harvesterRatio) /
FEE_PRECISION;
        if (_bounty > 0) {
          IERC20(_rewardToken).safeTransfer(_receiver, _bounty);
        }
        // @audit transfer rewards to operator
        IMultipleRewardDistributor(operator).depositReward(_rewardToken,
_rewardAmount - _incentive - _bounty);
      }
    }
  }
```

Status

The developer team has fixed this issue. Unclaimed rewards are now deducted when calculating the earnings in commit *8cc9d21*.

4.3.11 The `_operator` has not been granted authorization to transfer the `_rewardToken` tokens from the current contract.

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

Location

ConvexCurveManagerImmutable.sol

ConvexCurveManager.sol

Description

The `operator` contract does not have authorization for the current contract's `_rewardToken`, which will cause the transfer operation of `_rewardToken` in the `depositReward()` function to fail. Consequently, the `harvest()` function cannot be executed either.

```
function harvest(address _receiver) external {
   ......
        if (_bounty > 0) {
          IERC20(_rewardToken).safeTransfer(_receiver, _bounty);
        }

        IMultipleRewardDistributor(operator).depositReward(_rewardToken,
_rewardAmount - _incentive - _bounty);
      }
    }
  }
```

Status

The developer team has fixed this issue by adding authorization in the `syncRewardToken()` function in commit *4b41ab2*.

4.3.12 Add value validation for the parameter `_unvested` to save gas.

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

Location

ManageableVesting.sol

Description

The administrator can call the `cancel()` function to withdraw tokens the current user has not yet released. The parameter `_unvested` represents the number of tokens that the user has not yet released. When this value is 0, the following code can be skipped to save gas.

```
function cancle(address _user, uint256 _index) external override
onlyRole(DEFAULT_ADMIN_ROLE) {
    VestState memory _state = vesting[_user][_index];
    if (_state.cancelTime > 0) revert ErrorVestingAlreadyCancelled();

    uint32 _nowTime = uint32(block.timestamp);
    uint256 _vestedAmount = _getVested(_state, _nowTime);

    // @audit add a check for `_unvested`.
    uint256 _unvested = _state.vestingAmount - _vestedAmount;
    _state.cancelTime = _nowTime;
    vesting[_user][_index] = _state;

    // withdraw unvested to token to admin
    VestingManagerProxy(proxy[_user]).execute(
      managers[_state.managerIndex],
      abi.encodeCall(IVestingManager.withdraw, (_unvested, _msgSender()))
    );

    emit Cancle(_user, _index, _unvested, block.timestamp);
  }
```

Suggestion

When performing the `cancel()` operation, the `_unvested` parameter value should be greater than 0.

```
function cancle(address _user, uint256 _index) external override
onlyRole(DEFAULT_ADMIN_ROLE) {
    VestState memory _state = vesting[_user][_index];
    if (_state.cancelTime > 0) revert ErrorVestingAlreadyCancelled();

    uint32 _nowTime = uint32(block.timestamp);
    uint256 _vestedAmount = _getVested(_state, _nowTime);

    uint256 _unvested = _state.vestingAmount - _vestedAmount;

    //@audit add the check
    require(_unvested > 0,"all vested");

    _state.cancelTime = _nowTime;
    vesting[_user][_index] = _state;

    // withdraw unvested to token to admin
    VestingManagerProxy(proxy[_user]).execute(
      managers[_state.managerIndex],
      abi.encodeCall(IVestingManager.withdraw, (_unvested, _msgSender()))
    );

    emit Cancle(_user, _index, _unvested, block.timestamp);
  }
```

Status

The developer team fixed this issue in commit *8a6aeb9*. In this commit, they addressed the problem by adding a condition: when the parameter `_unvested` is 0, the function now directly returns without further processing.

4.3.13 Users cannot directly withdraw cvxFXN tokens or sdFXN tokens.

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

Location

CvxFxnVestingManager.sol

SdFxnVestingManager.sol

Description

According to the current code logic, after the expiration, users retrieve `stkCvxFxn` tokens or `sdFXN-gauge` tokens instead of directly retrieving `cvxFXN` or `sdFXN` tokens. It contradicts the statement in the design document, which states, "Users can claim underlying stake rewards after conversion, and only `cvxFXN` or `sdFXN` tokens can be withdrawn upon maturity." This design aspect needs to be reviewed for alignment.

```
// @audit vxFxnVestingManager.sol
function withdraw(uint256 _amount, address _receiver) external {
    IERC20(CVXFXN_STAKING).safeTransfer(_receiver, _amount);
  }

// @audit SdFxnVestingManager.sol
function withdraw(uint256 _amount, address _receiver) external {
    IERC20(SDFXN_GAUGE).safeTransfer(_receiver, _amount);
}
```

Status

From a user's perspective, after withdrawing funds, they might choose to deposit them again into protocols like Convex or StakeDAO. To simplify the process and save users gas fees, directly sending them stkCvxFxn tokens or sdFXN-gauge tokens could be a practical solution.

4.3.14 Users may potentially lose incentives within the sdFXN gauge.

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

Location

SdFxnVestingManager.sol

Description

Users can convert their vested FXN tokens into cvxFXN or sdFXN tokens and stake them in the Convex or StakeDao protocols. When a user chooses to deposit in the StakeDao protocol, they will call the `manage()` function in the `SdFxnVestingManager` contract. This action will deposit the unreleased FXN tokens into the `FXNDepositor` contract and convert them into sdFXN tokens before staking them into the corresponding gauge. Eventually, the `VestingManagerProxy` contract managed by the user will receive sdFXN-gauge tokens as share certificates.

It's important to note that due to the `incentiveToken`, the actual quantity of sdFXN-gauge tokens received by the user might not equal the amount of FXN tokens they deposited.

In the current code design, as long as either `_incentive > 0` or `_incentivePercentage > 0` conditions are met, FXN tokens will be directly deposited into the locker contract. If `incentiveToken > 0`, the actual amount of sdFXN-gauge tokens received will be greater than the amount of FXN tokens they deposited. However, during the claim process, the distribution is still based on the number of FXN tokens. It means that under the `VestingManagerProxy` contract, the remaining amount of `incentiveToken` in sdFXN-gauge tokens cannot be claimed, resulting in the user losing this portion of the funds.

```
function manage(uint256 _amount, address _receiver) external {
    address _redirect =
ICurveGauge(SDFXN_GAUGE).rewards_receiver(address(this));
    if (_redirect != _receiver) {
      ICurveGauge(SDFXN_GAUGE).set_rewards_receiver(_receiver);
    }

    IERC20(FXN).safeApprove(FXN_DEPOSITOR, 0);
    IERC20(FXN).safeApprove(FXN_DEPOSITOR, _amount);
    uint256 _incentive =
IStakeDAOSdTokenDepositor(FXN_DEPOSITOR).incentiveToken();
    uint256 _incentivePercentage =
IStakeDAOSdTokenDepositor(FXN_DEPOSITOR).lockIncentivePercent();
    if (_incentive > 0 || _incentivePercentage > 0) {
      //@audit users will incur partial principal loss.
      IStakeDAOSdTokenDepositor(FXN_DEPOSITOR).deposit(_amount, true,
true, address(this));
    } else {
      // @audit users will not incur any principal loss.
      IStakeDAOSdTokenDepositor(FXN_DEPOSITOR).deposit(_amount, false,
true, address(this));
    }
  }
```

```
// @audit
https://etherscan.io/address/0x7995192bE61EA0B28ce14183DDA51eDF78F1c7AB#c
ode#L768
function deposit(uint256 _amount, bool _lock, bool _stake, address _user)
public {
        if (_amount == 0) revert AMOUNT_ZERO();
        if (_user == address(0)) revert ADDRESS_ZERO();

        /// If _lock is true, lock tokens in the locker contract.
        if (_lock) {
            /// Transfer tokens to this contract
            IERC20(token).safeTransferFrom(msg.sender, address(locker),
_amount);

            /// Transfer the balance
            uint256 balance = IERC20(token).balanceOf(address(this));
            IERC20(token).safeTransfer(locker, balance);

            /// Lock the amount sent + balance of the contract.
            _lockToken(balance + _amount);

            /// If an incentive is available, add it to the amount.
            if (incentiveToken != 0) {
                //@audit adjust the user's FXN token quantity
                _amount += incentiveToken;

                emit IncentiveReceived(msg.sender, incentiveToken);

                incentiveToken = 0;
            }
        } else {
            /// Transfer tokens to the locker contract and lock them.
            IERC20(token).safeTransferFrom(msg.sender, address(this),
_amount);

            /// Compute call incentive and add to incentiveToken
            uint256 callIncentive = (_amount * lockIncentivePercent) /
DENOMINATOR;

            /// Subtract call incentive from _amount
            // @audit adjust the user's FXN token quantity
            _amount -= callIncentive;

            /// Add call incentive to incentiveToken
```

```
                incentiveToken += callIncentive;
            }

            if (_stake) {
                /// Mint sdToken to this contract.
                ITokenMinter(minter).mint(address(this), _amount);

                /// Deposit sdToken into gauge for _user.
                // @audit the actual number of sdFXN-gauge tokens received by
the user might not be equal to the quantity of FXN tokens they deposited.
                ILiquidityGauge(gauge).deposit(_amount, _user);
            } else {
                /// Mint sdToken to _user.
                ITokenMinter(minter).mint(_user, _amount);
            }
            emit Deposited(msg.sender, _user, _amount, _lock, _stake);
        }
```

Status

The developer team has acknowledged this issue. The amount of these incentives is generally tiny. Enabling the retrieval of these incentives would significantly increase the complexity of the contract design, leading to substantially higher gas fees. Therefore, the current code ensures that users in the vesting contract keep their principal, disregarding the negligible incentive amount. Additionally, the current contract utilizes the `delegatecall` method for invocation. If necessary, the code logic can be modified by replacing the contract.

4.3.15 Calling the `emaValue()` function may fail under specific conditions, consequently rendering critical functions of the f(x) protocol, such as mint, redeem, and liquidate, unusable.

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

Location

ExponentialMovingAverage.sol

Description

The function `emaValue()` calculates and updates the protocol's EMA value. After obtaining the intermediate variable `e`, the function checks the magnitude of the `e` value to prevent potential errors in subsequent mathematical operations. If the `e` value is less than *42139678854452767551* (approximately 42×10^{18}), the protocol's EMA value is updated. Subsequently, the contract uses the `LogExpMath.exp(para)` function to calculate the value of the variable `alpha`. Analyzing the parameter requirements of the `exp()` function reveals that $para \geq -41 \times 10^{18} \quad \&\& \quad para \leq 130 \times 10^{18}$. When the value of `e` falls within $e \in (41 \times 10^{18}, 42139678854452767551)$, the `LogExpMath.exp()` function will revert. In this case, critical functions of the f(x) protocol, such as mint, redeem, and liquidate, become unusable. Only after some time has passed and the condition $e \geq 42139678854452767551$ is satisfied, can the `f(x)` protocol resume normal functionality. However, this issue may recur periodically.

Essentially, *42139678854452767551* is derived from $\ln(0.5 \times 10^{-18}) \times 10^{18} = -42139678854452767551$, where the default minimum value is 0.5×10^{-18}. However, the `LogExpMath` library used in the current protocol, defaults to a minimum value of 1×10^{-18}, causing some `e` values to be handled incorrectly.

```
/// @dev Return the current ema value.
/// @param s The EMA storage.
function emaValue(EMAStorage memory s) internal view returns (uint256) {
  if (uint256(s.lastTime) < block.timestamp) {
    uint256 dt = block.timestamp - uint256(s.lastTime);
    uint256 e = (dt * PRECISION) / s.sampleInterval;
    if (e >= 42139678854452767551) {
      return s.lastValue;
    } else {
      uint256 alpha = uint256(LogExpMath.exp(-int256(e)));
      return (s.lastValue * (PRECISION - alpha) + s.lastEmaValue * alpha)
/ PRECISION;
    }
  } else {
    return s.lastEmaValue;
  }
}

// @audit LogExpMath.sol
library LogExpMath {
......
// The domain of natural exponentiation is bound by the word size and
number of decimals used.
//
```

```
// Because internally the result will be stored using 20 decimals, the
largest possible result is
// (2^255 - 1) / 10^20, which makes the largest exponent ln((2^255 - 1) /
10^20) = 130.700829182905140221.
// The smallest possible result is 10^(-18), which makes largest negative
argument
// ln(10^(-18)) = -41.446531673892822312.
// We use 130.0 and -41.0 to have some safety margin.
int256 constant MAX_NATURAL_EXPONENT = 130e18;
int256 constant MIN_NATURAL_EXPONENT = -41e18;
......

function exp(int256 x) internal pure returns (int256) {
    // @audit  x>= -41e18
    require(x >= MIN_NATURAL_EXPONENT && x <= MAX_NATURAL_EXPONENT,
"INVALID_EXPONENT");

    if (x < 0) {
      // We only handle positive exponents: e^(-x) is computed as 1 /
e^x. We can safely make x positive since it
      // fits in the signed 256 bit range (as it is larger than
MIN_NATURAL_EXPONENT).
      // Fixed point division requires multiplying by ONE_18.
      return ((ONE_18 * ONE_18) / exp(-x));
    }
   ......

}
```

Suggestion

Modify the condition `e >= 42139678854452767551` to `e >= 41e18`.

Status

The development team has adopted our suggestion and has fixed this issue in commit bbed30a.

5. Conclusion

After auditing and analyzing the f(x) Protocol New Features, 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.