Steadefi Audit Report

Prepared by: X3 Security

Table of Contents

Protocol Summary

Steadefi is the next-gen DeFi protocol designed to provide the highest and most sustainable real yields to our investors without the stress of constant position management or the prolonged downturns of the crypto markets. Steadefi provides vaults with automated risk management for earning leveraged yields effectively and passively in bull, crab, and bear markets. With lending and leveraged delta long and neutral strategies, Steadefi's vaults cater to different risk/reward strategies to the best yield-generating DeFi protocols.

Audit Period: October 26th, 2023 - November 6th, 2023

Disclaimer

The X3 Security team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Impact	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

Scope

```
contracts
├── interfaces
    ├── oracles
        ├── IChainlinkOracle.sol
        ├── IGMXOracle.sol
    ├── strategy
        ├── gmx
            ├── IGMXVault.sol
            ├── IGMXVaultEvent.sol
├── oracles
        ├── ChainlinkARBOracle.sol
        ├── GMXOracle.sol
├── strategy
        ├── gmx
            ├── GMXCallback.sol
            ├── GMXChecks.sol
            ├── GMXCompound.sol
            ├── GMXDeposit.sol
            ├── GMXEmergency.sol
            ├── GMXManager.sol
            ├── GMXProcessDeposit.sol
            ├── GMXProcessWithdraw.sol
            ├── GMXReader.sol
            ├── GMXRebalance.sol
            ├── GMXTrove.sol
            ├── GMXTypes.sol
            ├── GMXVault.sol
            ├── GMXWithdraw.sol
            ├── GMXWorker.sol
└── utils
    ├── Errors.sol
```

Roles

Lender: Lenders deposit assets to Lending Vaults (1 asset per Lending Vault) to earn safer, more stable borrow interest on their assets.

Depositor: Depositors deposit assets to Strategy Vaults (Strategy Vaults could accept different assets) to earn higher yields than if they were to supply their assets to the yield-earning protocol directly. Depending on the strategy, however, they take on different types of risk which would affect their final profit and losses.

Keeper: Keepers are automated "bots" that run 24/7, frequently scheduled and/or event-triggered code scripts to perform various protocol maintenance tasks. These tasks include updating of borrow interest rates for Lending Vaults, rebalancing Strategy Vaults whose health are out of its strategy parameter limits, compounding earned yield for Strategy Vaults, reverting certain issues for strategy vaults when they occur, and triggering Emergency Pauses for lending and strategy vaults in the event of any possible issues.

Owner: Owners are administrators that have rights to configure and update various sensitive vault configurations and parameters. Owners of deployed smart contracts (vaults, oracles, etc.) should be Timelocks which are managed by Multi-Sigs that require at least a 2 out of 3 signing approval for any transactions to happen with a 24-hour delay. Note that on contract deployment, the immediate Owner is the hot wallet deployer account. After deploying and initial configuration of the contract, the ownership should be immediately transferred from the hot wallet deployer to a Timelock managed by a Multi-Sig.

Issues found

Findings Table

Severity	ID	Title
Medium	M-01	[M-01] Lack of Negative Price Checks Due to Incorrect Operator in ChainlinkARBOracle
Medium	M-02	[M-02] ChainlinkARBOracle contract will return wrong price for assets if underlying aggregator hits minAnswer
Low	L-01	[L-01] Inability to Update or Remove Chainlink Price Feeds

[M-01] Lack of Negative Price Checks Due to Incorrect Operator in `ChainlinkARBOracle`

Severity: Medium Risk
Likelihood: Medium
Date Modified: Nov 4th, 2023

Relevant GitHub Link: Link to code

Summary
The ChainlinkARBOracle contract is designed to process and validate Chainlink oracle responses. However, its current implementation only checks if the response is zero and omits potential negative values. This oversight, attributed to the use of the `==` operator instead of `<=`, can introduce inaccuracies in the deviation calculation, potentially affecting key functions like `consult` and `consultIn18Decimals`.

Vulnerability Details
Within the ChainlinkARBOracle contract, the `_badChainlinkResponse` function is tasked with validating Chainlink oracle responses:

```
// Check for non-positive price
if (response.answer == 0) { return true; }
```

This approach exclusively identifies zero responses, neglecting possible negative outcomes. The implications of this narrow check are most evident in functions like `_badPriceDeviation`. Since the value `answer` is an `int256` variable it might return a negative value if the price of that token drops drastically, for example, when oil futures dropped below zero.

Impact

- **Deviation Miscalculations**: In scenarios where `currentResponse.answer` holds a negative value contrasting a positive `prevResponse.answer`, for instance, with a `currentResponse.answer` of -5 and a `prevResponse.answer` of 10, the deviation calculation in `_badPriceDeviation` will not yield the intended results, leading to potential miscalculations. The resulting computation may not align with the contract's expected behavior.
- **Inaccurate Function Outputs**: The outputs of functions, specifically `consult`, could be compromised by such miscalculations. This might return negative price data and incorrect decimals.
- **Risk of Function Reverts**: Functions like `consultIn18Decimals`, which rely on these verifications, may inadvertently revert if presented with unexpected negative price data. This poses operational risks for the contract and any other contract that is calling this function due to the fact that this function will revert when a negative value is provided as function `toUint256()` from SafeCast reverts it to prevent underflow/overflow.

Tools Used
Manual Code Review

Recommendations

Revise Chainlink Response Verification

To bolster the integrity of the contract, it's recommended to modify the validation in the `_badChainlinkResponse` function to encompass all non-positive responses:

```
// Suggested update to _badChainlinkResponse
if (response.answer <= 0) { return true; }
```

By implementing this refined check, the ChainlinkARBOracle contract can more accurately and consistently handle Chainlink oracle data, reducing the potential risks stemming from unanticipated price feed values.

[M-02] `ChainlinkARBOracle` contract will return wrong price for assets if underlying aggregator hits minAnswer

Severity: Medium Risk
Likelihood: High
Date Modified: Nov 4th, 2023

Relevant GitHub Link: Link to code

Summary

Chainlink aggregators have a built-in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value, the price of the oracle will continue to return the minPrice instead of the actual price of the asset. This would allow users to continue borrowing with the asset but at the wrong price. This is exactly what happened to Venus on BSC when LUNA imploded.

Vulnerability Details

ChainlinkAggregators have minAnswer and maxAnswer circuit breakers built into them. This means that if the price of the asset drops below the minAnswer, the protocol will continue to value the token at minAnswer instead of its actual value. This will allow users to take out huge amounts of bad debt and bankrupt the protocol.

The function `_getChainlinkResponse()` is used to get the price of the requested token:

```
function _getChainlinkResponse(address _feed) internal view returns
(ChainlinkResponse memory) {
    ChainlinkResponse memory _chainlinkResponse;

    _chainlinkResponse.decimals = AggregatorV3Interface(_feed).decimals();

    // Arbitrum sequencer uptime feed
    (
      /* uint80 _roundID*/,
      int256 _answer,
      uint256 _startedAt,
```

```
        /* uint256 _updatedAt */,
        /* uint80 _answeredInRound */
      ) = sequencerUptimeFeed.latestRoundData();

      // Answer == 0: Sequencer is up
      // Answer == 1: Sequencer is down
      bool _isSequencerUp = _answer == 0;
      if (!_isSequencerUp) revert Errors.SequencerDown();

      // Make sure the grace period has passed after the
      // sequencer is back up.
      uint256 _timeSinceUp = block.timestamp - _startedAt;
      if (_timeSinceUp <= SEQUENCER_GRACE_PERIOD_TIME) revert
  Errors.GracePeriodNotOver();

      (
        uint80 _latestRoundId,
        int256 _latestAnswer,
        /* uint256 _startedAt */,
        uint256 _latestTimestamp,
        /* uint80 _answeredInRound */
      ) = AggregatorV3Interface(_feed).latestRoundData();

      _chainlinkResponse.roundId = _latestRoundId;
      _chainlinkResponse.answer = _latestAnswer;
      _chainlinkResponse.timestamp = _latestTimestamp;
      _chainlinkResponse.success = true;

      return _chainlinkResponse;
  }
```

However, there are no checks in place if an asset price falls below minAnswer. The `latestRoundData` extracts the linked aggregator and requests round data from it. If an asset's price falls below the minAnswer, the protocol continues to value the token at the minAnswer rather than its real value. This discrepancy could have the protocol end up minting drastically larger amounts of assets as well as returning a much bigger collateral factor.

For example, if the minAnswer for TokenA is set at 1 dollar and its actual price drops to 0.10 dollars, the aggregator persists in reporting a value of $1. This results in the associated function calls recognizing a value that's tenfold greater than TokenA's real worth.

It's important to note that while Chainlink oracles form part of the OracleAggregator system and the use of a combination of oracles could potentially prevent such a situation, there's still a risk. Secondary oracles, such as Band, could potentially be exploited by a malicious user who can DDOS relayers to prevent price updates. Once the price becomes stale, the Chainlink oracle's price would be the sole reference, posing a significant risk.

Impact
In the event of an asset crash (like LUNA), the protocol can be manipulated to handle calls at an inflated price.

Tools Used
Manual Review

Recommendations
The function `_getChainlinkResponse` should cross-check the returned answer against the minAnswer/maxAnswer and revert if the answer is outside of these bounds:

```
(
    uint80 _latestRoundId,
    int256 _latestAnswer,
    /* uint256 _startedAt */,
    uint256 _latestTimestamp,
    /* uint80 _answeredInRound */
) = AggregatorV3Interface(_feed).latestRoundData();

if (_latestAnswer >= maxAnswer || _latestAnswer <= minAnswer) revert();
_chainlinkResponse.roundId = _latestRoundId;
_chainlinkResponse.answer = _latestAnswer;
_chainlinkResponse.timestamp = _latestTimestamp;
_chainlinkResponse.success = true;

return _chainlinkResponse;
```

[L-01] Inability to Update or Remove Chainlink Price Feeds

Severity: Low Risk
Likelihood: Low
Date Modified: Nov 4th, 2023

Relevant GitHub Link: Link to code Link to code

Summary
The ChainlinkARBOracle contract provides a function to set Chainlink price feeds for tokens. However, once a feed is set, there's no mechanism to update or remove it. This design restricts flexibility and adaptability in changing circumstances.

Vulnerability Details
The `addTokenPriceFeed` function in the ChainlinkARBOracle contract allows the owner to set a Chainlink price feed for a specific token. The function checks if a feed for the given token is already set and reverts if it is. This means, post the initial setting, the feed cannot be modified or removed.

```
if (feeds[token] != address(0)) revert Errors.TokenPriceFeedAlreadySet();
```

This limitation can be problematic if the set feed becomes unreliable, is deprecated, or

if there's a need to switch to a different, more reliable source.

Impact

- **Inflexibility**: The contract might become stuck with outdated or unreliable data sources.
- **Need for Contract Redeployment**: If a feed needs an update, the entire contract might need redeployment, leading to potential migration challenges.
- **Redundant Data**: Without the ability to remove, deprecated tokens' data remain in the contract, leading to inefficiencies.

Tools Used
Manual Review

Recommendations
To address the above concerns, it is recommended:

Implementing an Update Function: This function will allow the contract owner to update the feed for a given token. Here's a potential implementation:

```
function updateTokenPriceFeed(address token, address newFeed) external onlyOwner {
    require(token != address(0), "ZeroAddressNotAllowed");
    require(newFeed != address(0), "ZeroAddressNotAllowed");
    require(feeds[token] != address(0), "TokenPriceFeedNotSet");

    feeds[token] = newFeed;
}
```

Implementing a Removal Function: This function will allow the owner to remove a feed for a token, setting its address back to the zero address:

```
function removeTokenPriceFeed(address token) external onlyOwner {
    require(token != address(0), "ZeroAddressNotAllowed");
    require(feeds[token] != address(0), "TokenPriceFeedNotSet");

    delete feeds[token];
}
```