OpenZeppelin | security

TideFlow DAO Audit

April 17, 2024

Table of Contents

Summary

Type	DeFi
Timeline	From 2024-03-11 To 2024-04-01
Languages	Solidity

Total Issues	21 (20 resolved)
Critical Severity Issues	1 (1 resolved)
High Severity Issues	1 (0 resolved)
Medium Severity Issues	2 (2 resolved)
Low Severity Issues	5 (5 resolved)
Notes & Additional Information	12 (12 resolved)
Client Reported Issues	0 (0 resolved)

Scope

We audited the `TideFlow-DAO/tideflow-contracts` repository at the `484379a696e33c6478845767b0eff9910a80834e` commit.

In scope were the following files:

```
contracts
├── EpochAdvancer.sol
├── Governed.sol
├── OwnableERC20.sol
├── PoolFactory.sol
├── TideFlow.sol
├── TideFlowEvents.sol
├── TideFlowLoupe.sol
├── TideFlowUUPSUpgradeable.sol
├── interfaces
│   ├── AggregatorV3Interface.sol
│   ├── IAccountingModel.sol
│   ├── IAnchorRateModel.sol
│   ├── IFactory.sol
│   ├── IOwnableERC20.sol
│   ├── IPriceOracle.sol
│   └── ITideFlow.sol
├── models
│   ├── AccountingModel.sol
│   ├── AnchorRateModel.sol
│   └── AnchorRateModelV3.sol
├── oracles
│   ├── ChainlinkOracle.sol
│   └── ChainlinkOracleReverse.sol
└── storage
    ├── StorageGoverned.sol
    └── StorageTideFlow.sol
```

System Overview

The TideFlow contracts introduce a system featuring two distinct types of participants: anchors and surfers. Anchors represent a category of users who prioritize safeguarding their investment's value with a lower risk profile. On the other hand, surfers cater to users who have a higher tolerance for risk, in pursuit of potentially greater rewards. These contracts are upgradeable and governed by a DAO.

Surfers supply liquidity by depositing the underlying pool token and assume risk from anchors. Anchors, through depositing the underlying token, secure their investment against price declines up to a specified price limit. Both users will receive an ERC-20 token representing their proof of liquidity.

The system operates in epochs, with profit calculations and system state transitions at each epoch's end. Epoch length varies across pools and is initially configured upon creating new pools. In the event the price of the underlying pool token increases, the surfers will benefit from some of the profits made by the anchors up until a calculated `upsideExposureRate`. In the event the price of the underlying pool token decreases, the anchors are protected up to a calculated `downsideProtectionRate` using the underlying pool tokens of the surfers.

Depositing in a Pool

Users looking to deposit in a pool need to choose between two roles, anchor or surfer, before the epoch begins by depositing the pool's underlying token. Surfers will use the `depositSurfer` function, while anchors will use `depositAnchor`. It is important to note that once a user commits to a role, this choice is irreversible and they can only increase the size of their deposit up until the targeted epoch is finalized.

After the epoch starts, both anchors and surfers can redeem their `anchorTokens` and `surferTokens`, based on their deposits and the tokens' fluctuating prices in each epoch.

Switching Sides in a Pool

Users able to redeem their anchor or surfer tokens can switch roles, from anchor to surfer or vice versa. For simplicity, we describe switching from surfer to anchor, though the reverse

process is the same. Users can switch sides using `redeemAndSwitchSideSurferTokens` if they have not redeemed their surfer tokens, or `switchSideSurfer` if they have. These functions signal the user's intent to switch, allowing them to claim their new anchor tokens after the epoch ends using the method `redeemAnchorTokensSwitchSide`.

Exiting a Pool

To exit a pool and retrieve the underlying assets, users must indicate their intent to leave. Taking the exit from the surfer's perspective for illustration, though the process for anchors is identical, there are two methods to signal an exit: `redeemAndExitSurferTokens` if the user has not yet redeemed their surfer tokens, or `exitSurfer` if they have. After signaling, users can redeem their underlying assets once the current epoch ends using the method `redeemSurferUnderlying`.

Security Model and Trust Assumptions

The system's security is heavily dependent on the DAO, thus, it is presumed that the DAO effectively handles the setting of oracles, along with the accounting and anchor rate models. For the system to function as designed, certain trust assumptions are crucial and must remain valid so long as the roles with special privileges are upheld. Deviations from these assumptions could lead to adverse effects:

- The system uses Chainlink oracles for asset price retrieval. Chainlink's data feeds, which the oracle utilizes for price information, are deemed reliable.
- Prices from oracles are converted to `uint256`; this conversion process is assumed to be secure.
- The system is incompatible with specific types of ERC-20 tokens, such as those that rebase or have fees on transfers.

Privileged Roles

The following actors are considered to have privileged roles within the system since they can perform special actions that can influence the end-users. Here is a summarized list:

- The `guardian`, which is a multi-sig wallet owned by the TideFlow team, can:
  - Pause and unpause the system.
  - Transfer the `guardian` role.
- The `DAO`, which is implemented by a governance contract (out of scope), can:
  - Pause and unpause the system.
  - Transfer the `guardian` role.
  - Transfer the `DAO` role.
  - Set the price oracle.
  - Set the anchor rate model, which is responsible for calculating the `downsideProtectionRate` and `upsideExposureRate`.
  - Set the accounting model, which is responsible for calculating the surfer and anchor profits.
  - Set the fee recipient address.
  - Set the percentage of the fees applied, up to a maximum of 100% of the surfer or anchor profits.
  - Upgrade the system at any time to another version, changing the underlying logic.
- The `PoolFactory` owner can:
  - Set the implementation of the TideFlow pools
  - Deploy new TideFlow pools

Critical Severity

C-01 Liquidity Calculation Error in `advanceEpoch` Function

When advancing an epoch, the contract updates variables that account for liquidity, considering four factors:

1. Liquidity entering the system through user deposits, as accounted for in `queuedSurfersUnderlyingIn` and `queuedAnchorsUnderlyingIn`
2. Liquidity exiting the system through redemptions by users, tracked in `queuedSurferTokensBurn` and `queuedAnchorTokensBurn`
3. Liquidity switching sides, either from anchor to surfer or vice versa, indicated by `queuedAnchorTokensSwitchSide` and `queuedSurferTokensSwitchSide`
4. Liquidity from profits

The correct approach for calculating the total liquidity in the upcoming epoch should account for the inflow from deposits, switches and profits, and the outflow from redemptions, switches and losses. However, the `advanceEpoch` function currently only adds the liquidity entering from switches to the receiving side without subtracting it from the departing side. For example, while the surfer's underlying tokens departing from switches are recorded in `queuedAnchorsUnderlyingIn`, it is removed from `surferUnderlyingOut`, and thus not subtracted from the surfer's side in the calculation, leading to inaccurate liquidity accounting.

This inaccuracy in calculating available liquidity for the next epoch critically impacts the protocol. It influences the rate calculations and the number of tokens minted for users. By assuming it possesses more liquidity than actually available, the contract might falsely conclude it can protect anchor investments, potentially leading to a solvency problem and overestimating profits. Additionally, solvency issues arise from minting liquidity tokens under the incorrect assumption of sufficient liquidity. These solvency problems may result in a scenario where initial users can withdraw their funds, while subsequent users cannot.

Consider fixing the calculations within the `advanceEpoch` function to accurately reflect liquidity deductions associated with users switching between surfers and anchors.

Update: *Resolved in pull request #6. The TideFlow team stated:*

> *We refactored the advance epoch a bit to remove switch side functionality piggybacking on the entry/exit queues processing.*

High Severity

H-01 Lockup Risk in the `advanceEpoch` Function

The `advanceEpoch` function is triggered each time a non-view function of the contract is called. Consequently, a revert in this function could result in the contract being locked. Potential causes for such a revert include arithmetic underflow or overflow, or a division by zero. As an epoch ends and a new one begins, the function undertakes several calculations, including the computation of the prices for both surfer and anchor tokens. A division by zero could occur within the internal functions `_processSurferQueues` or `_processAnchorQueues` for either token type, since the token price, which serves as the divisor for calculating the quantity of surfer or anchor tokens based on the amount of the underlying asset, could be zero.

If we consider the case of a failure in `_processSurferQueues`, the price of the surfer token is calculated using the formula `($.epochSurferLiquidity * scaleFactor) / supply`. While precautions are taken against a zero supply, if the supply is larger than `$.epochSurferLiquidity * scaleFactor`, the resulting division could round down to zero. This situation could occur, for instance, if the price of the underlying asset drops over several epochs consecutively, leading surfers to absorb the losses, which in turn reduces the `epochSurferLiquidity`.

To prevent scenarios where the contract remains locked and users are unable to withdraw their underlying assets, one option could be to add an emergency function that can only be invoked by the DAO to unlock the trapped funds with the optional ability to reset the variables in the pool to their default values. It is crucial to ensure that this function can only be called if the contract is locked. It is also worth mentioning that relying on extensive functions performing numerous calculations, such as `advanceEpoch`, is generally not considered best practice and is prone to errors.

Update: *Acknowledged, not resolved. The TideFlow team stated:*

> *We acknowledge the fact that certain extreme edge cases could potentially cause the `advanceEpoch` function to fail. We assess the risk of the described scenario occurring to be sufficiently low. Apart from that, the system demonstrates a satisfactory level of*

> *resilience during computation. The DAO will need to consider such a use case when determining which pools to launch. Furthermore, while addressing other issues identified in this audit, we have modified the behavior of the price oracle to explicitly fail if the returned price is 0. In such an instance, it is imperative for the DAO to review and actively manage the situation. At this juncture, we opt not to introduce any emergency function. Instead, we emphasize that the system is upgradeable, enabling the DAO to respond in such circumstances.*

Medium Severity

M-01 Missing Oracle Result Validation

The protocol integrates with Chainlink oracles in order to fetch the price of the configured asset. However, when fetching the price using `oracle.latestRoundData()`, the result is not validated. For instance:

- The price might return zero, which could lead to losses for the surfers.
- The oracle might return a stale price, which could lead to unfair distribution of the underlying funds given the current market price.

Consider validating both `answer` and `updatedAt` of the `oracle.latestRoundData()` output to ensure the oracle returned a recent and correct price.

Update: *Resolved in pull request #8.*

While the fix itself is correct, please make sure to select a proper value of `maxPriceAge` depending on the oracle set by the pool.

There are two triggers which kick off Chainlink nodes to update the price in their contracts:

- Price deviation: if the price deviates past some interval, the price on-chain will be updated.
- Time interval: if the price stays within the deviation parameter, it will update every X minutes/hours.

For popular data feeds, like ETH/USD, the time interval is set to 1 hour, so this should be good. However, for less common oracles, like KNC/ETH (looking at a random one from here), the time interval is 24 hours. This means if the price did not change with this 2% deviation interval, it could be that the nodes are not updated for 24 hours. Therefore, if `maxPriceAge` is set to

`3600`, `getPrice` will revert if the asset hasn't been updated in the last hour (which is plausible for these data feeds).

M-02 Missing Decimals Validation

The DAO can modify the oracle with the `setPriceOracle` function. This function verifies that the new oracle address points to a contract containing code but fails to check for consistency in the number of decimals compared to the old oracle. This oversight can lead to incorrect calculations if the new feed uses a different amount of decimals. Since the pool's calculations depend on a specific decimal amount, discrepancies can result in financial losses for the users and the protocol.

Consider adding checks to prevent decimal mismatches in the `setPriceOracle` function.

Update: *Resolved in pull request #9.*

Low Severity

L-01 Missing Docstrings

Throughout the codebase, there are multiple instances of code that does not have docstrings.

Consider thoroughly documenting all variables and functions (and their parameters) that are part of any contract's public API. Especially the important functions and variables such as:

- The `redeemAllSurferTokens` function in `TideFlow.sol`.
- The `redeemAllAnchorTokens` function in `TideFlow.sol`.
- The `scaleFactor` state variable in `TideFlow.sol`.
- The `calcSurferProfits` function in `AccountingModel.sol`.

Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #7.*

L-02 PoolFactory Constructor Sets Owner Twice

When deploying the `PoolFactory` contract, the constructor is setting the owner twice. Once during the calling of the `Ownable` constructor and once by calling `transferOwnership`. Moreover, if the passed `owner` address is not equal to the deployer, the call to `transferOwnership` will revert as `transferOwnership` can only be called by the owner.

Consider removing the call to `transferOwnership`, as the owner is already set by calling `Ownable(owner)`.

Update: *Resolved in pull request #3.*

L-03 Pragma Statement Spans Several Minor Versions of Solidity

Using a pragma statement that spans several minor Solidity versions can lead to unpredictable behavior due to differences in features, bug fixes, deprecation, and compatibility between minor versions.

Within `AggregatorV3Interface.sol`, there is a `pragma` statement that spans several minor versions of Solidity.

Consider pinning the Solidity version more specifically throughout the codebase to ensure predictable behavior and maintain compatibility across various compilers.

Update: *Resolved in pull request #4.*

L-04 Next Epoch Estimation Does Not Take Into Account Switchers

The function `estimateNextEpoch` estimates the `surferLiquidity`, `anchorLiquidity`, `upsideRate` and `downsideRate` for the next epoch based on the current conditions. However, the estimations for `surferLiquidity` and `anchorLiqudity` do not take into account surfers switching to anchors and vice versa. For example if a surfer switches side to anchor, this liquidity should be removed from the `surferLiquidity` and added to the `anchorLiquidity`. As the liquidities are used for calculating both the `downsideProtectionRate` and `upsideExposureRate`, these estimations might be inaccurate as well.

Consider taking into account the surfers and anchors switching sides when estimating the `surferLiquidity` and `anchorLiquidity` to reflect a more accurate state of the pool.

Update: *Resolved in pull request #5.*

L-05 Possible Overflow in `calcSurferProfits` Function

The `calcSurferProfits` function calculates surfer profits at the end of an epoch using the formula `(x * y * totalAnchors) / (currentPrice * scaleFactor)`. Here, `x` is defined as the difference between the current price and the entry price (`currentPrice - entryPrice`), and `y` equals the difference between the scale factor and the upside exposure rate (`scaleFactor - upsideExposureRate`).

In certain edge cases, the product of these three components (`x * y * totalAnchors`) may overflow. The variable `x`, which represents the change in prices, can be particularly volatile. Given that the asset being tracked could experience significant price fluctuations, the epoch's duration might lead to `x` reaching values as high as ten times the entry price or more. This is especially possible for low-priced coins, with the price feed being ETH/coin or BTC/coin, represented in 18 decimals, indicating a very large value.

The `totalAnchors` variable represents the total underlying value of the anchors, which could theoretically be capped at 100 million USD of tokens. It is possible to choose a low value per wei token, considering that the team mentioned the underlying pool token might differ from the token whose price the pool is tracking.

Considering all these points, the values in the calculation can easily approach the maximum value for a `uint256`. However, these scenarios are considered edge cases and are not expected to occur under normal circumstances. It's suggested to ensure these scenarios are unattainable to prevent overflow issues.

Update: *Resolved in pull request #21.*

Notes & Additional Information

N-01 Constants Not Using `UPPER_CASE` Format

Throughout the codebase there are constants not declared using `UPPER_CASE` format. For instance:

- All constants declared in `AnchorRateModel.sol`.
- All constants declared in `AnchorRateModelV3.sol`.
- The `scaleFactor` constant declared in `TideFlow.sol`, `TideFlowLoupe.sol`, and `AccountingModel.sol`.
- The storage locations in `StorageGoverned.sol` and `StorageTideFlow.sol`.

According to the Solidity Style Guide, constants should be named with all capital letters with underscores separating words. For better readability, consider following this convention.

Update: *Resolved in pull request #10. The TideFlow team stated:*

> *We also removed `AnchorRateModelV3` since it is no longer needed.*

N-02 File Specifies an Outdated Solidity Version

Within `AggregatorV3Interface.sol`, there is a `pragma` statement that uses an outdated version of Solidity.

Consider taking advantage of the latest Solidity version to improve the overall readability and security of the codebase. Regardless of which version of Solidity is used, consider pinning the version consistently throughout the codebase to prevent bugs due to incompatible future releases.

Update: *Resolved in pull request #11.*

N-03 Incomplete Docstrings

Throughout the codebase, there are several instances of incomplete docstrings. For instance:

- The parameters `entryPrice`, `currentPrice`, `downsideProtectionRate`, `totalAnchors` and `totalBalance` of the `calcAnchorProfits` function in `AccountingModel.sol` are not documented.
- The return value of the `feesAccrued` function in `TideFlow.sol` is not documented.

Consider thoroughly documenting all functions/events (and their parameters or return values) that are part of a contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #12.*

N-04 Missing Named Parameters in Mappings

Since Solidity 0.8.18, developers can utilize named parameters in mappings. This means mappings can take the form of `mapping(KeyType KeyName? => ValueType ValueName?)`. This updated syntax provides a more transparent representation of the mapping's purpose.

Within `StorageTideFlow.sol`, there are multiple mappings without named parameters. For instance:

- The `history_epochSurferTokenPrice` state variable.
- The `history_epochAnchorTokenPrice` state variable.
- The `surferEntryQueue` state variable.
- The `surferExitQueue` state variable.
- The `surferSwitchSideQueue` state variable.
- The `anchorEntryQueue` state variable.
- The `anchorExitQueue` state variable.
- The `anchorSwitchSideQueue` state variable.

Consider adding named parameters to the mappings to improve the readability and maintainability of the code.

Update: *Resolved in pull request #13.*

N-05 Unnecessary Casts

Within `TideFlowLoupe.sol`, there are unnecessary casts. For instance:

- The `IERC20(tf.surferToken())` cast as `tf.surferToken()` already returns an `IERC20`.
- The `IERC20(tf.anchorToken())` cast as `tf.anchorToken()` already returns an `IERC20`.

To improve the overall clarity, intent, and readability of the codebase, consider removing unnecessary casts.

Update: *Resolved in pull request #14.*

N-06 Misleading Comments

The following misleading and inconsistent comments have been identified in the codebase:

- According to this comment, the `enforceCallerGuardian` enforces a call comes from the Guardian. However, it also accepts calls from the DAO address.

Consider revising the comments to improve consistency and more accurately reflect the implemented logic.

Update: *Resolved in pull request #15.*

N-07 Gas Inefficiencies

While auditing the contracts, we spotted some opportunities to save gas:

- In `advanceEpoch`, the functions `getCurrentAnchorProfits` and `getCurrentSurferProfits` are both fetching the latest price from the Chainlink oracle. Additionally, the price is fetched a third time in `advanceEpoch`.

Consider reducing the amount of times the price is fetched from the Chainlink oracle when calling `advanceEpoch`. This reduces both the amount of external calls as well as the fees to be paid to Chainlink for consuming the oracle.

When performing these changes, aim to reach an optimal tradeoff between gas optimization and readability. Having a codebase that is easy to understand reduces the chance of errors in the future and improves transparency for the community.

Update: *Resolved in pull request #16.*

N-08 Using `int/uint` Instead of `int256/uint256`

Throughout the codebase, there are uses of `int/uint`, as opposed to `int256/uint256`. For instance:

- On line 17 of `ChainlinkOracle.sol`.
- On line 199 of `TideFlowLoupe.sol`.

In favor of explicitness, consider replacing all instances of `int/uint` with `int256/uint256`.

Update: *Resolved in pull request #17.*

N-09 Unused Imports

Throughout the codebase, there are imports that are unused and could be removed. For instance:

- The import `import {OwnableERC20} from "../OwnableERC20.sol";` imports unused alias `OwnableERC20` in `StorageGoverned.sol`.
- The import `import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";` imports unused alias `IERC20` in `StorageGoverned.sol`.
- The import `import {IPriceOracle} from "./interfaces/IPriceOracle.sol";` imports unused alias `IPriceOracle` in `TideFlow.sol`.
- The import `import {IAnchorRateModel} from "./interfaces/IAnchorRateModel.sol";` imports unused alias `IAnchorRateModel` in `TideFlow.sol`.

Consider removing unused imports to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #18.*

N-10 State Variable Visibility Not Explicitly Declared

Throughout the codebase, there are state variables that lack an explicitly declared visibility. For instance:

- The `splitPoint` state variable in `AnchorRateModel.sol`.
- The `maxProtectionPercentage` state variable in `AnchorRateModel.sol`.
- The `maxProtectionAbsolute` state variable in `AnchorRateModel.sol`.
- The `splitPoint` state variable in `AnchorRateModelV3.sol`.
- The `maxProtectionPercentage` state variable in `AnchorRateModelV3.sol`.
- The `maxProtectionAbsolute` state variable in `AnchorRateModelV3.sol`.

For clarity, consider always explicitly declaring the visibility of variables, even when the default visibility matches the intended visibility.

Update: *Resolved in pull request #19.*

N-11 Protocol Fee Can Be Set To 100% Of The Profits

Upon finalizing an epoch and calculating the profits, the system incurs a protocol fee. The protocol fee is configured as a percentage of either the surfers' profits (if the price increased) or the anchors' profits (if the price decreased). While the protocol fee is initially set to 0%, the fee can be changed by the DAO using the `setFeesPercentage` function up to a maximum fee percentage. Currently, this maximum fee percentage is set to 100%. In case the fee percentage is set to 100% by the DAO, all profits from either the surfers or anchors are converted to protocol fees.

Consider reducing the maximum fee percentage to a reasonable percentage of the surfer and anchor profits.

Update: *Resolved in pull request #20.*

N-12 Normalization of Decimal Places in `ChainlinkOracleReverse` Contract

The Chainlink reverse oracle contract is designed to invert price feeds, for instance, it converts an `ETH/BTC` price feed into `BTC/ETH`. However, a limitation of this contract is that it

consistently returns prices with 8 decimal places, regardless of whether the original price was denoted in 18 decimals.

Consider documenting this behavior and explicitly state that the reverse oracle contract normalizes price feeds to 8 decimal places to prevent misunderstandings and future issues.

Update: *Resolved in pull request #9*

Conclusion

During this three-week engagement, the audit team conducted an in-depth examination of the codebase, employing fuzzing techniques, and discovered a critical issue as well as several lower severity findings. While the code itself was clear and well-documented, the documentation might benefit from some additional insights on the anchor rate models and how they succeed in incentivizing both surfers and anchors depending on the amount of liquidity in the protocol.

Additionally, although the codebase features an extensive suite of unit tests, this does not guarantee the code's safety. The critical vulnerability found could have been easily detected by a test. We also recommend exploring more advanced testing techniques, such as fuzzing and invariant testing. These types of tests encompass a broader range of cases and ensure the system meets specific conditions under a set of defined assumptions, potentially uncovering critical flaws.

Communication with the team has been excellent, with all questions from the auditors being answered promptly and comprehensively.