MCDEX Mai Fund Protocol Audit

OPENZEPPELIN SECURITY | OCTOBER 9, 2020

Security Audits

Monte Carlo Decentralized Exchange is a decentralized derivatives exchange. The Mai Fund Protocol is a trading tool built on top of the Mai Protocol. It allows users to purchase shares in "funds", which then trades perpetual futures on their behalf. Funds can be managed by a fund manager which manually conducts trades, or by a predefined trading strategy. In this audit, we reviewed the smart contracts of the Mai Fund Protocol. The audited commit is `98af0d1d7e9872ba2b5e734e2a43c161ef635608`, and the scope includes all production contracts within the `contracts` directory **except** those within the `contracts/test` directory, `Migrations.sol` and `Context.sol`. All external code and contract dependencies were assumed to work as documented.

Here we present our findings.

Summary

Overall, we found the code to be well-constructed, using encapsulated functions and isolated contracts to keep the design modular. We found that the Monte Carlo team has paid special attention to avoid common issues such as reentrancy.

Many of the identified issues stemmed from an inconsistent coding style, such as inconsistent use of SafeMath, or incorrect conversion between internal accounting units and actual collateral units. Issues also stemmed from mistakes in complicated data structure management or from a lack of input sanitization.

all other unrelated changes to the code base.

System overview

A user is able to create a fund to trade leveraged `LONG` or `SHORT` positions within the Mai Protocol. Then, users are able to purchase shares in such funds by depositing collateral. They are able to redeem those shares to receive collateral back. A fund can be controlled manually by a `Manager`, which receives fees in return for operating the fund, or by a predetermined trading algorithm stored in a smart contract. Publicly callable functions exist in automatically traded funds which allow anyone to trigger rebalances of the fund's positions. Such users are referred to as `Keepers`, and they receive an incentive to purchase positions from the fund.

In certain cases, a fund can be shut down. Such cases include: when the fund's shares have lost enough of their value relative to their previous maximum value, when the fund's leverage is too high, when the Mai Protocol experiences an emergency shutdown, or when the fund `Manager` chooses to shut down the fund. When a fund is shut down, it first enters an `Emergency` state, then, after all of the fund's positions are closed, it can enter a `Shutdown` state wherein users can withdraw their funds.

Privileged roles

Many of the contracts have privileged roles that can significantly affect the usefulness and safety of the system. For instance:

- The `owner` is the owner of the Mai Protocol. This account can pause a fund, or change fund parameters like fees, the max number of mintable shares, or the `_drawdownHighWaterMark` and `_leverageHighWaterMark` parameters.
- The `Manager` role receives fees from the fund's operation.

Ecosystem dependencies

As the ecosystem becomes more interconnected, understanding external dependencies and assumptions has become an increasingly crucial component of system security. To this end, we would like to discuss how the financial contracts depend on the surrounding ecosystem.

During this audit, we assumed that the Chainlink oracle will be available and reliable, and will update fast enough to reflect real-time market conditions.

Additionally, the Mai Fund protocol relies on the `Keeper` role to take actions to shut down at-risk funds before they become insolvent. If this is not done in time, funds may become disabled. For this audit, we assume that the `Keeper` is willing and able to perform protective actions for at-risk funds.

Critical severity

[C01] Miscalculation of payout in `settle`

Within `SettleableFund.settle()`, the variable `collateralSettled` is calculated as `_rawBalanceOf(_self()) - _totalFeeClaimed`. However, the values `_rawBalanceOf(_self())` and `_totalFeeClaimed` are not expressed in the same terms.

`_rawBalanceOf` returns `_toRawAmount(X)`, where `X` is either the balance of `account` in terms of the underlying ERC20 token units or wei. When it is called within `settle()`, the value that is passed in will be the address of the `SettleableFund` contract. `_toRawAmount` will then divide this value by `_scaler` and return. So, the value given back is in terms of `collateralUnits / _scaler`.

On the other hand, `_totalFeeClaimed` is represented in different terms. When the fee is withdrawn by the admins within `SocialTradingFund.sol`, while `_totalFeeClaimed` is decreased by `collateralAmount`, the amount that is pushed to the user will be `_toRawAmount(collateralAmount)`. `_toRawAmount`, as we have seen above, divides `collateralAmount` by `_scaler` and returns this. So, `_totalFeeClaimed` is accounted in terms of `collateralUnits * _scaler`.

Clearly, when subtracting these values, we are assuming they are in comparable terms. Really, what happens in the code is that they are NOT in the same terms. `_rawBalanceOf` is in terms

The potential implications of this are twofold. In one case, `_totalFeeClaimed` will be much higher than `_rawBalanceOf(_self())`, since it is `_scaler ** 2` orders of magnitude greater. This will cause the `.sub()` to revert, rendering the `settle()` function useless. The other potential outcome is that `collateralSettled` will be much less than it should be, since `_rawBalanceOf(_self())` will be much lower than it should be. This will translate to `collateralToReturn` being lower than it should be. Although `shareAmount` of user's shares in the fund will be burned, the amount that is pushed to the user will be `_toRawAmount(collateralToReturn)`, which will divide the already-too-low `collateralToReturn` value by `_scaler` once again, and then transfer that many token units or wei to the user.

Since the identified location is the only place in which `_rawBalanceOf` is called, consider changing the logic of `_rawBalanceOf` to better suit that application. Instead of calling `_toRawAmount`, which divides the balance of tokens or wei by `_scaler`, consider calling `_toInternalAmount`, which multiplies by `_scaler`. Additionally, consider changing the name of `_rawBalanceOf` to better describe what it does, perhaps renaming it to `_internalBalanceOf`.

Update: *Fixed in PR #3.*

High severity

None.

Medium severity

[M01] Function `abs` breaks for minimum `int256`

Within the `LibMathEx` library, the `abs` function simply negates any negative values passed in.

For unsigned integers in twos complement format, there exists one extra negative value than positive value. So, for the minimum value of `int256`, there exists no complementary positive value that can be stored in an `int256`. When this value is passed into the `abs` function, the same negative value will be returned, which should never happen with an absolute value.

Update: *Fixed in PR #4.*

[M02] Lack of SafeMath

In the `LibEnumerableMap` library, there are many instances where unprotected math operators are used. For example:

- line 40 uses `-` operator
- line 82 uses `-` operator
- line 91 uses `-` operator
- line 92 uses `+` and `/` operators
- line 111 uses `-` operator

Consider using `SafeMath`'s corresponding functions instead of unprotected mathematical operators.

Update: *Fixed in PR #5.*

[M03] Unsafe divisions in `LibMathEx`

The function `wdiv` on line 19 and function `wdiv` on line 32 of the `LibMathEx` library, accepts the divisor `y` as an input parameter. However, these functions do not check if the value of `y` is zero.

If the value passed for `y` is `0`, the division will revert due to the division by zero. To prevent such unsafe calculations, consider using the `div` functions provided in OpenZeppelin's `SignedSafeMath` and `SafeMath` libraries, instead of direct division via the `/` operator.

Update: *Fixed in PR #6.*

[M04] No check on `newTimestamp` value from `_priceFeeder`

Within the `updatePrice` function in the `PeriodicPriceBucket` contract, the values of `newPrice` and `newTimestamp` are returned when `priceFeeder.price()` is called. While the system checks for an invalid value of `newPrice`, there is no check for invalid value of `newTimestamp`.

of `newTimestamp` might be `0`. This will result in `periodIndex` being `0`.

To avoid this edge case, consider adding a check which requires that `newTimestamp > 0`.

Update: *Fixed in PR #7.*

[M05] In certain edge case, funds can get locked

In the case that the `marginBalance` of a fund ends up being less than 0, the `totalAssetValue()` function will revert. If this happens, many functions which rely on `_totalAssetValue()` will be locked, resulting in reverts whenever they are called. Some of these functions are:

- `BaseFund.purchase()`
- `BaseFund.redeem()`
- `BaseFund.bidRedeemingShare()`
- `Getter.netASsetValue()`
- `Getter.netAssetValuePerShare()`
- `Getter.leverage()`
- `Getter.drawdown()`
- `SettleableFund._canShutdown()`
- `AutoTradingFund.rebalanceTarget()`
- `SocialTradingFund.managementFee()`
- `SocialTradingFund._withdrawManagementFee()`
- `SocialTradingFund.updateManagementFee()`
- `SettleableFund.setShutdown()`

Some notable side effects are that users will not be able to withdraw or shut the fund down.

However, this is an edge case which relies on the fund not being liquidated within the Mai Protocol, and the fund not being shut down after the `_drawdownHighWaterMark` has been reached. Note that both of these actions are intended behaviors that should happen far before the fund's `marginBalance` goes negative.

prevent locking funds should this state ever be reached.

Update: *Acknowledged. The MonteCarlo team states that the maintenance of the margin account is one of the most important mechanisms of Mai Protocol V2. A user's margin account would be liquidated before its value could go below maintenance margin, which is always greater than 0.*

[M06] Unchecked address used to instantiate contract

In the `initialize` function of the `AutoTradingFund` contract, the value of `strategyAddress` address is input by the user. This address is later used to instantiate the `ITradingStrategy` interface. However, there is no validation in the system to check if the `strategyAddress` address is an address of a contract or a zero address.

If an incorrect address is used to instantiate the `ITradingStrategy` interface, the calls to the `_nextTargetLeverage` function will revert which in-turn affects the rebalancing of the fund's positions.

Additionally, the `perpetualAddress` address is input by the user in the `initialize` function of the `AutoTradingFund` contract and the `initialize` function of the `SocialTradingFund` contract. These functions in turn call the `__SettleableFund_init` function in the `SettleableFund` contract, which then calls the `__MarginAccount_init_unchained` function in the `MarginAccount` contract. Within the `__MarginAccount_init_unchained` function, the value of `perpetualAddress` address is used to instantiate the `IPerpetual` interface. Although, this function checks that the `perpetualAddress` address should not be a zero address, there is no validation that `perpetualAddress` address is a contract and not an Externally Owned Account address.

Similarly, the `_setPriceFeeder` function in the `PeriodicPriceBucket` contract takes `newPriceFeeder` address as an input, checks if the address is not a zero address and instantiates the `IPriceFeeder` interface. This function also fails to validate if `newPriceFeeder` address is a contract address. Another example of the same is the `priceSeriesRetriever` address in the `RSIReader` contract which instantiates the `IPriceSeriesRetriever` interface.

[illegible] addresses are indeed contracts by using the [illegible] function provided in the OpenZeppelin `contracts` Address library.

Update: *Fixed in PR #8.*

Low severity

[L01] Inaccessible elements can be added to `_transfers`

Within the `RSITrendingStrategy` contract, the `constructor` sets values in the `_transfers` mapping. The indices for both dimensions of each element must be less than or equal to `maxSegment`. `maxSegment` is set to the length of `_seperators + 1`. Thus, the max value for either index of `_transfers` is `_seperators.length + 1`.

The only place where `_transfers` is accessed is within `getNextTarget`. There, `segment` must be less than or equal to `_seperators.length`, and `_lastSegment` must be a valid segment (meaning that it is also equal to or less than `_seperators.length`). If it is not, `_lastSegment` is set to `segment`'s value. In all cases, `_lastSegment` and `segment` can never equal `maxSegment`, meaning that any elements in `_transfers` which have `maxSegment` as an index will never be used.

Consider changing the condition on lines 62 and 63 of `RSITrendingStrategy` contract to be `<` instead of `<=`.

Update: *Fixed in PR #9.*

[L02] In RSITrendingStrategy, `_transfers` and `_seperators` can be set only once

Within the `RSITrendingStrategy` contract, the values of `_seperators` and `_transfers` are set in the `constructor`. Once set, these values cannot be changed. Furthermore, since `_transfers` is set within a `for` loop, there is a non-negligible chance that calling the constructor with too large of a `transferEntries` array could cause execution to run out of gas and revert.

For example, in the [illegible] function of [illegible] contract, a value of `0` may be interpreted as the `nextTargetLeverage`. This may happen if the user creating `RSITrendingStrategy` leaves some value out of `transferEntries`, by mistake or to save gas.

Consider creating some methods to change `_transfers` after the constructor has been called, perhaps also adding a flag which prevents the `getNextTarget` method from being successfully called until `_transfers` has been fully set and committed. Alternatively, consider informing users of `RSITrendingStrategy` that there is a limit to the number of `_transfer` elements which can be set, and that unset values will be interpreted as valid `0` values. Otherwise, errors or misconceptions about the code may be unchangeable and result in unexpected behavior.

Update: *Fixed in PR #10. The Monte Carlo team has decided to make the strategy replaceable.*

[L03] `rebalanceTarget` may return undefined values

Within the `AutoTradingFund` contract, the `rebalanceTarget` function will return `0` for `amount` and `Side.FLAT` for `side` if `needRebalance` is false, as the values for `amount` and `side` are only set within the `if` branch where `needRebalance` is true.

Within the audited codebase, `rebalanceTarget` is only called from one place, and it is immediately followed by a `require` that `needRebalance == true`. So, in the case that `needRebalance` is `false`, execution will revert and state will not be affected.

In case future use of `needRebalance` is desired, consider implementing some default value for `amount` and `side`, and making a note of this in comments above the function. Otherwise, the function may behave unexpectedly for future developers.

Update: *Fixed in PR #11.*

[L04] `addBucket` has no upper bound on input

The function `addBucket` within the `PeriodicPriceBucket` contract adds a value to the the `_periods` enumerable set. If this value is too high, when it is later accessed and used to calculate `periodIndex`, `periodIndex` may evaluate to `0`, causing

Consider adding an upper bound on `period` within `addBucket` such that for reasonable values of `newTimestamp`, `periodIndex` will be greater than 0.

Update: *Fixed in PR #12.*

[L05] `_seperators` values not checked

Within the `RSITrendingStrategy` contract, the array `_seperators` is set within the `constructor`, but the values within it are never checked.

Although there is a comment indicating that `_seperators` should be monotonically increasing, it is easily possible for it not to be. If a value at index `n+1` is less than the value at index `n`, that value of `_seperators` will never result in usage and is effectively wasted space. Additionally, the max possible value of `_seperators` is not limited, but since it is compared to `rsi`, it is pointless to have any values greater than the max value of `rsi`.

Consider checking the values of `_seperators` to ensure that they are monotonically increasing and that they do not exceed `RSI_UPPERBOUND` (the max value for `rsi`). Alternatively, consider adding to the comment on line 36 that `_seperators` should never be greater than `RSI_UPPERBOUND`, and making this clear in the documentation as well.

Update: *Fixed in PR #13.*

[L06] Misleading, incomplete or missing docstrings

Although most of the functions in the codebase have relevant docstrings, there are some instances where the docstrings are misleading, incomplete or missing. For example:

- The docstrings above the `rebalanceTarget` function in the `AutoTradingFund` contract says that the function `Return true if rebalance is needed`. It does not mention anything about the remaining two return variables `amount` and `side`.

- The docstrings above the `_performanceFee` function in the `Fee` contract implies that there are two parts of the performance fee calculation, when in reality there is only one.

Consider changing the identified misleading or incomplete docstrings to better reflect the code's behavior. Additionally, consider adding docstrings wherever relevant. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Fixed in PR #11 and PR #14.*

[L07] `wmul` and `wdiv` round incorrectly in some cases

Within the `LibMathEx` library, the `wdiv` function adds half of the divisor to the dividend before dividing by it. This is done to assist in rounding, since division natively truncates (rounds down). By adding half, any divisions which in conventional mathematics should round up, will do so in this code.

However, this function always adds half of the divisor, even when the dividend and divisor have opposite signs. This decreases the absolute value of the result, when it should be increasing it, and the end result is that the returned value is less accurate than it should be. For instance, when calling this function with the values `(-200, 40)` or `(100, -40)`, the results are `-4.999...e18` and `-2.4999...e18` respectively, when they should be `-5e18` and `-2.5e18`. This is especially noteworthy given that inputs of `(200, 40)` and `(100, 40)` result in `5e18` and `2.5e18`.

Similarly, for the `wmul` function, when the signs of `x` and `y` differ, the answer will be inaccurate. This can be seen by calling this function with input `(5e18, 0.5e18)`, and comparing this result to that of the input `(-5e18, 0.5e18)` or `(5e18, -0.5e18)`.

Solidity will round towards 0 when truncating. To counteract the effect of adding two numbers of different sign, consider adding logic to the `wdiv` function such that when `x` and `y` are opposite signs, the `.add` on line 32 is replaced by a `.sub`. For the `wmul` function, very similar logic should be implemented, such that the `.add` on line 28 should be replaced by `.sub` only when the signs of `x` and `y` differ. Various combinations of positive and negative values should be tested to ensure correct functionality.

Update: *Fixed in PR #15.*

example, there are instances where the returned variables are named and not declared within the function, such as in the `wmul` function in `LibMathEx` library and there are instances where the return variables are unnamed, such as in the `set` function in `LibEnumerableMap` library.

Consider removing all named return variables and explicitly declaring them as local variables where needed. This should improve both explicitness and readability of the project.

Update: *Acknowledged. The Monte Carlo team has decided to keep the code as-is for the current version.*

[L09] Unnecessary frequent reads from storage in loops

Within the function `buckets` in the `PeriodicPriceBucket` contract, the for loop checks the value of `_periods.length()` on each iteration. Since this function is already potentially gas-intensive (due to looping) consider accessing `bucketCount` for the looping condition. This may help to prevent out-of-gas errors and will improve gas efficiency.

Similarly, the for loop inside the `updatePrice` function uses `_periods.length()` in its looping condition. Consider declaring a memory object, setting it to `_periods.length()`, and using that value in the looping condition instead of `_periods.length()`.

Update: *Fixed in PR #16.*

[L10] Outdated solidity version used in the `IAggregator` contract

An outdated and unpinned version of solidity is used in the `IAggregator` contract.

Consider pinning the version of the Solidity compiler to its latest stable version. This should help prevent introducing unexpected bugs due to incompatible future releases. To choose a specific version, developers should consider both the compiler's features needed by the project and the list of known bugs associated with each Solidity compiler version.

Update: *Fixed in PR #23.*

[L11] Unfixed version of `contracts-ethereum-package`

To protect against unexpected changes that may affect the code, consider pinning a specific version of `openzeppelin-contracts-ethereum-package`.

Update: *Fixed in PR #18.*

[L12] Unnecessary return variables

In the codebase, there are instances where functions return an unnecessary value.

For example, in the `Collateral` contract, the function `_pullFromUser` returns a variable `rawAmount`. However, the value of this variable is passed as an input to the function, thus making the return variable unnecessary.

Similarly, the `_pushToUser` function in the `Collateral` contract returns the `rawAmount` variable unnecessarily.

In order to reduce the surface for error and reduce developer confusion, please consider removing any unused return variables.

Update: *Fixed in PR #19. Additionally, an unused return value from the `_updateFee` function has been removed.*

[L13] It is not possible to redeem right when the locking period ends

The `_canRedeem` function in the `ERC20CappedRedeemable` contract implements a check which ensures that, after purchasing the shares, users wait for a specific time period before they can redeem those shares. The system ensures that as soon as the current timestamp is greater than the wait period, the user is allowed to redeem. However, this conditional check misses the scenario where the current timestamp can be equal to the wait period.

Consider allowing users to redeem as soon as the wait period is reached, updating the `<` comparison operator to `<=`. Alternatively, if this is by design, consider explicitly documenting that in order to reedeem, users will have to wait until the block after the locking period ends.

Update: *Fixed in PR #20.*

parameter. However, the implementation of this interface, which is in the `mai-protocol-v2` repository, accepts `address broker` as an input parameter.

Although `Mai Protocol V2` is out-of-scope for this audit, the team identified that this mismatch also exists in the `removeBroker` function of the `IGlobalConfig` interface in the `mai-protocol-v2` repo. The latest commit of the `mai-protocol-v2` project at the time of this audit is `090dcd0c7980760a7b85eafc449dbb7164bae32d`.

Consider adding the `(address broker)` as an input parameter to the `removeBroker` function of the `IGlobalConfig` interface for both `mai-fund-protocol` and `mai-protocol-v2` projects.

Update: *Fixed in PR #21 for `mai-fund-protocol` project and PR #14 for the `mai-protocol-v2` project.*

[L15] Zero address can be set as the fund manager

The `setManager` function in the `SocialTradingFund` contract allows the owner to set a fund manager. However, this function does not check if the input `newManager` address is a zero address.

This scenario does not break the code since the check in the `_withdrawManagementFee` function prevents the payment of management fee to a zero address and the owner can always change the manager from zero address to a valid address by calling the `setManager` function. However, in order to remove any confusion, consider putting a check in `setManager` function to ensure that the value of `newManager` is not a zero address.

Update: *Fixed in PR #22.*

[L16] Function `__BaseFund_init` is marked internal but is never called

The `__BaseFund_init` function in the `BaseFund` contract is marked as `internal` but this function is not called anywhere within the codebase.

Since this is an `initializer` function, consider either calling it within the codebase or mark the function `external`.

Notes & Additional Information

[N01] Commented out code

In `RSITrendingStrategy` contract, lines 49-59 contain what appears to be old code which has been commented out.

Other instances where commented code is present in the codebase are:

- Lines 101-104 of the `Getter` contract
- Lines 32-33 of the `BaseFund` contract

If the code is no longer needed, and the comments do not pertain to the currently implemented code, the commented-out code should be removed to minimize confusion for future developers and auditors. Consider removing the commented-out code.

Update: *Fixed in PR #24*.

[N02] `_entranceFee` calculation unnecessarily complex

Within the `_entranceFee` function, a portion of the input value `purchasedAssetValue` is returned, which represents the entrance fee given that value.

While calculating the entrance fee, the calculation is not what one may expect. Instead of the calculation being `purchasedAssetValue * _entranceFeeRate`, the calculation is actually `purchasedAssetValue * _entranceFeeRate / (1 + _entranceFeeRate)`. This means that when `_entranceFeeRate` is set to 10%, the `_entranceFee` calculation will return `0.1/1.1 = 9.09%` of `purchasedAssetValue`.

This is confusing, since `_streamingFee` is calculated using `_streamingFeeRate` as a percentage per year, such that over 1 year, if `_streamingFeeRate` represents 10% per year, then the `_streamingFee` will be 10% of the input `netAssetValue`. Similarly, `_performanceFee` will be calculated as `_performanceFeeRate` percentage of the difference between `netAssetValue` and `_maxNetAssetValue`.

be set arbitrarily, its value can be adjusted to achieve the same result as the original calculation, but this change will make the calculation more understandable for users and developers.

Update: *Acknowledged. The Monte Carlo team has decided to keep the formula as it is, however, they have updated the docstrings above the* `_entranceFee` *function and have changed variable names in PR #25 for the purpose of clarity.*

[N03] Implicit casting

Within `Collateral.sol`, the `uint8` `decimals` is implicitly cast to `uint256`, as the `sub` function expects two `uint256` values.

To favor readability, consider explicitly casting the value `decimals` to `uint256`.

Update: *Fixed in PR #26.*

[N04] Redundant `require` statements

Within the `LibMathEx` contract, on line 23 and line 36 there are `require` statements which prevent `z == 0`. However, on line 24 and line 37, `.div(z)` is performed. SafeMath's `div` function and SignedSafeMath's `div` function both contain checks that the second parameter is not `0`. Therefore, the `require`s within `LibMathEx.sol` are redundant and unneeded.

Consider removing the identified `requires` to simplify the code.

Update: *Fixed in PR #40.*

[N05] Naming issues

Several places in the code could benefit from clearer naming. Here are our suggestions:

- `_maxNetAssetValuePerShare` should be renamed to `_historicMaxNetAssetValuePerShare`. `_maxNetAssetValue` and `_updateMaxNetAssetValuePerShare` should be renamed similarly.
- `_totalFeeClaimed` should be renamed to `_totalFeeClaimable`, since it decreases when fees are withdrawn.

Update: *Fixed in PR #27.* `_maxNetAssetValue` *was not renamed.*

[N06] `require` checks modify state

The `require` checks on line 90 of the `addBucket` and line 102 of the `removeBucket` functions in the `PeriodicPriceBucket` contract modifies the state of `_periods` variable.

To make the code cleaner and easier to understand, consider moving these calls out of the `require` checks, and instead assigning the return value to some variable, then using this variable in the `require` check.

Update: *Fixed in PR #28.*

[N07] Unnecessary operation in `_managementFee`

Within the function `_managementFee` in the `Core` contract, the second assignment to `assetValue` is not needed, as `assetValue` is not used afterwards in the function and is locally scoped.

Consider removing the specified operation. This will make the code cleaner and easier to understand.

Update: *Fixed in PR #30.*

[N08] `IAggregator` interface name is not consistent

Within `IAggregator.sol`, the name of the interface, `AggregatorInterface`, does not match the name of the contract file, `IAggregator`.

In order to improve readability and to be consistent with the naming of other interfaces in the codebase, consider changing the name of `AggregatorInterface` interface to `IAggregator`.

Update: *Fixed in PR #29.*

[N09] Inconsistent capitalization in `enums`

Consider capitalizing the identifiers in the `FundState` enum to increase consistency within the codebase.

Update: *Fixed in PR #31.*

[N10] `LibTypes` is unused as a library

`LibTypes` is imported as a library and is used to access `LibTypes.Side` in the `AutoTradingFund` contract. However, instead of being used as a library, it is called directly on line 143. To use it as a library, it should be called as a function of `targetSide`, which would instead read as `targetSide.opposite()`.

Since this is the only place where any function from `LibTypes` is called, consider changing line 143 as described to use `LibTypes` as a library, or consider removing line 32 to make the code simpler and easier to understand.

Update: *Fixed in PR #32.*

[N11] `_toInternalAmount()` is unused

The `internal` function `_toInternalAmount()` is unused in the audited codebase.

Consider removing it to simplify the code and reduce confusion. Alternatively, if it was intended to make this function `public`, consider making that change. Or, if it was intended to be called within `SettleableFund.settle()`, consider making the change described in **[C01] Miscalculation of payout in `settle`**.

Update: *Fixed in PR #3. The `_toInternalAmount()` function is now used as described in C01.*

[N12] Unused events

There are events in the codebase that are never emitted. For instance:

- `IncreaseWithdrawableCollateral` and `DecreaseWithdrawableCollateral` events in the `ERC20CappedRedeemable`

Consider either removing the declaration or emitting the event appropriately.

Update: *Fixed in PR #33.*

[N13] Unused variable

Within `RSIReader` contract, in the `_calculateRSI` function, variable `lastNonZeroPrice` is declared and set many times, but it is not used anywhere within the function. Notably it is declared locally, so it cannot be used outside the scope of this function.

Consider removing the unused variable to make the code cleaner and easier to understand.

Update: *Fixed in PR #34.*

[N14] Unused constant

The `constant` `SHARE_TOKEN_DECIMALS` declared in the `LibConstant` library is not used within the codebase.

Consider removing it to make the code cleaner.

Update: *Fixed in PR #35.*

[N15] Unused import statements

Consider removing the following unused imports:

- `EnumerableMap` and `Arrays` in `LibEnumerableMap.sol`
- `Math` and `LibConstant` in `RSIReader.sol`
- `Context` in `State.sol`
- `LibConstant` in `BaseFund.sol`

Update: *Fixed in PR #36.*

[N16] Typos

- On line 148 of `BaseFund.sol`, “wiil” should be “will”.
- On line 150 of `BaseFund.sol`, “shares token” should be “share tokens”.
- On line 259 of `BaseFund.sol`, “Trding” should be “Trading”.
- On line 273 of `BaseFund.sol`, “excceeded” should be “exceeded”.
- On line 13 of `Collateral.sol`, “underlaying” should be “underlying”.
- On line 54 of `ERC20CappedRedeemable.sol`, “share will still belongs” should be “share will still belong”.
- On line 80 of `ERC20CappedRedeemable.sol`, “Slipage” should be “Slippage”.
- On line 153 of `ERC20CappedRedeemable.sol`, “&&” should be “&” or “and”.
- On line 163 of `ERC20CappedRedeemable.sol`, “receipient” should be “recipient”.
- On line 165 of `ERC20CappedRedeemable.sol`, “a account continously purchase && transfer shares to another account.” should be “an account continuously purchases and transfers shares to another account”.
- On line 166 of `ERC20CappedRedeemable.sol`, “unexpeced” should be “unexpected”.
- On line 33 of `Fee.sol`, line 42 of `Fee.sol` and line 51 of `Fee.sol` “rete” should be “rate”.
- On line 37 of `Fee.sol`, line 46 of `Fee.sol`, and line 55 of `Fee.sol`, “too large rate” should be “rate too large”.
- On line 80 of `Fee.sol`, “365 day” should be “365 days”.
- On line 50 of `PeriodicPriceBucket.sol`, “period” should be “periods”.
- On line 52 of `PeriodicPriceBucket.sol`, “remove” should be “removal”.
- On line 53 of `PeriodicPriceBucket.sol`, “period” should be “periods”.
- On line 69 of `PeriodicPriceBucket.sol`, “a period is exist” should be “if a period exists”.
- On line 70 of `PeriodicPriceBucket.sol`, “is already existed” should be “already exists”.
- On line 89 of `PeriodicPriceBucket.sol`, “reaches limit” should be “has already reached the limit”.
- On line 102 of `PeriodicPriceBucket.sol`, “is not” should be “does not”.
- On line 134 of `PeriodicPriceBucket.sol`, “overrided” should be “overwritten”.
- On line 161 of `PeriodicPriceBucket.sol`, “is not” should be “does not”.
- On line 23 of `RSITrendingStrategy.sol`, “implements” should be “implementations”.

- On line 37 of `RSITrendingStrategy.sol` and line 81 of `RSITrendingStrategy.sol`, “ouput” should be “output”.
- On line 79 of `SettleableFund.sol`, “drawdonw” should be “drawdown”.
- On line 104 of `SettleableFund.sol`, “leveraga” should be “leverage”.
- On line 48 of `State.sol`, “shutdown state” should be “Emergency state”.

Consider correcting these typos to improve code readability.

Update: *Fixed in PR #37.*

[N17] Misleading error message

On line 160 of `PeriodicPriceBucket` contract, the error message implies `beginTimestamp` must be less than `endTimestamp`, when the check allows them to also be equal. The error message should be updated to say “begin must be earlier than or equal to end”.

Consider updating the identified error message to better reflect the code’s behavior.

Update: *Fixed in PR #38.*

Conclusions

1 critical and no high severity issues were found. Some changes were proposed to follow best practices, reduce the potential attack surface, and enhance overall code quality.

Update: *We reviewed the fixes applied by the Monte Carlo team and all of the issues have been fixed or acknowledged.*

Related Posts

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits