MCDEX Mai Protocol Audit

OPENZEPPELIN SECURITY | JUNE 5, 2020

Security Audits

Monte Carlo Decentralized Exchange is a decentralized derivatives exchange. Mai Protocol V2 is a system that allows users to trade trust-minimized Perpetual Futures contracts on the Ethereum blockchain. Users can either trade with each other or trade against the Automated Market Maker smart contract. In this audit, we reviewed the smart contracts within this system. The audited commit is `4b198083ec4ae2d6851e101fc44ea333eaa3cd92` and the scope includes all production contracts in the `contracts` directory. The contracts in the `test` directory were not included.

Additionally, it should be noted that the system design includes a number of economic arguments and assumptions. These were explored to the extent that they clarified the intention of the code base, but we did not audit the mechanism design itself.

All external code and contract dependencies were assumed to work as documented.

Update: *All issues listed below have been fixed or accepted by the Monte Carlo team. In contrast to our recommendation, many fixes were committed simultaneously alongside code refactors, which made it harder to consider the issues and their mitigations in isolation. Instead, we reviewed the expected code segments in the final commit for each issue, and disregarded all other changes to the code base.*

Here we present our findings, along with the updates for each of them.

Summary

as well as recognizing the risks associated with the privileged roles in the system.

Our main concern is the lack of function-level documentation and a noticeable absence of comments, particularly for a project of this complexity. Although the external documentation was very helpful, we believe introducing comprehensive function-level documentation throughout the code base will make it significantly easier to understand, reason about, maintain and update.

Update: *The Monte Carlo team have greatly improved the comments and documentation.*

System overview

Users of the Mai Protocol V2 can open leveraged positions that track the price of the supported asset pairs. After depositing sufficient collateral, they are able to trade `LONG` or `SHORT` positions that will benefit from increasing or decreasing prices of the underlying asset respectively. These positions can also be leveraged to increase volatility.

In the event a user becomes over-leveraged, their position can be liquidated by any other trader. The liquidator effectively buys the over-leveraged positions and the liquidated position incurs a penalty that is partly sent to the liquidator and partly sent to the global insurance fund. If a liquidated user is unable to cover their debt, the insurance fund covers the difference. If the insurance fund is empty, further losses get socialized among the traders who hold the opposite position.

One interesting component of the system is the Automated Market Maker (AMM) contract, which holds `LONG` positions and collateral. Liquidity providers can deposit additional collateral as well as `LONG` contracts (technically they sell the AMM `LONG` contracts and retain rights over those positions) to increase the depth of the market. In exchange, they will receive ERC20 tokens corresponding to their share of the liquidity pool. Users of the AMM contribute fees to the pool, which means the token holders receive trading fees until they withdraw their liquidity. Naturally, since they are ERC20 tokens, the liquidity providers can also trade these tokens directly.

Once the AMM is funded, any user can trade `LONG` positions with this contract at deterministically generated prices. Whenever the AMM price deviates appreciably from the oracle price, collateral is transferred from `LONG` to `SHORT` position holders or vice versa (whether or not they obtained

perpetual contracts.

To avoid nullifying trades that were submitted to the order book, traders cannot simultaneously trade through the order book and the AMM. They can choose either option whenever desired, but there is a short delay in the transition where their account may not be fully functional.

Update: *The trading limitation and delay have been removed.*

Privileged roles

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

- the `PerpetualProxy` contract has privileged access to make deposits, withdrawals and trades on behalf of any user. Naturally, the smart contract itself cannot abuse this power but any malicious user with the same role could. The system should be configured to prevent this possibility.
- the `Perpetual` administrator has control over all of the governance parameters, including the leverage amount, the fees and all penalty rates.
- the `Perpetual` administrator can update the contract that manages safety time locks, can set the address that receives development fees, or can even replace the entire AMM.
- the `Perpetual` administrator can withdraw directly from the insurance fund.
- the `Perpetual` administrator can put the contract in an emergency settlement mode, where they can set the price of a futures contract and adjust all user balances.
- the `AMM` administrator can control the funding rate parameters and the fee rates.

Users of the protocol will need to trust the administrators to use these powers fairly and wisely.

Ecosystem dependencies

As the ecosystem becomes more interconnected, understanding the 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.

Most importantly, the AMM uses Chainlink to obtain the price of the underlying asset. If this dependency becomes corrupted, it could be used to steal funds within the system. In some cases

Additionally, the system uses time-based logic to calculate the AMM funding. Very long delays between updates to the funding calculation (for instance, during periods of high Ethereum congestion) would increase the gas costs of the subsequent updates, potentially to a prohibitive degree.

Critical severity

[C01] Anyone can liquidate on behalf of another account

The `Perpetual` contract has a public `liquidateFrom` function that bypasses the checks in the `liquidate` function.

This means that it can be called to liquidate a position when the contract is in the `SETTLED` state. Additionally, any user can set an arbitrary `from` address, causing a third-party user to confiscate the under-collateralized trader's position. This means that any trader can unilaterally rearrange another account's position. They could also liquidate on behalf of the Perpetual Proxy, which could break some of the Automated Market Maker invariants, such as the condition that it only holds `LONG` positions.

Consider restricting `liquidateFrom` to `internal` visibility.

Update: *Fixed. The `liquidateFrom` function has been removed.*

[C02] Orders cannot be cancelled

When a user or broker calls `cancelOrder`, the `cancelled` mapping is updated, but this has no subsequent effects. In particular, `validateOrderParam` does not check if the order has been cancelled.

Consider adding this check to the order validation to ensure cancelled orders cannot be filled.

Update: *Fixed. The validation now checks the cancellation status.*

High severity

holders should cover the loss. In this way, the profits on one side are garnished to fund the loss on the other side. This ensures the system as a whole cannot become insolvent. However, the loss is actually attributed to positions on the same side. In the worst case, none of the positions on the same side will be able to cover the loss, which means the contract will be underfunded and some profits will not be redeemable. Consider updating the code to assign losses to the opposite side of the liquidation.

Update: *Fixed.*

Medium severity

[M01] `liquidateFrom` does not use `tradingLotSize`

In the `Perpetual` contract, within the function `liquidateFrom` there is a check that the `maxAmount` to be liquidated is a multiple of `lotSize`. This may be the amount of some user's position that gets liquidated by the end of the function call.

However, all trades which happen during a call to `matchOrders` must be for an amount that is a multiple of `tradingLotSize`. This is the only way to trade with other users. Otherwise, trading with the AMM is also limited through checks in `buyFrom` and `sellFrom` such that the user can only trade amounts that are a multiple of `tradingLotSize`.

In the `PerpetualGovernance` contract, we can see that `tradingLotSize` must be a multiple of `lotSize`. If these two numbers differ, and a user is liquidated for an amount not divisible by `tradingLotSize`, they may have a left-over position that is not an even multiple of `tradingLotSize`. If this occurs, they may not be able to close their position completely.

Consider restricting liquidation amounts to multiples of `tradingLotSize`, rather than `lotSize`. Alternatively, consider providing a method for liquidated traders to close their positions that are of less than `tradingLotSize`.

Update: *Acknowledged. Typically,* `lotSize` *and* `tradingLotSize` *will be the same value, but* `tradingLotSize` *might be increased to help with high Ethereum congestion.*

However, the liquidation price and position side is inferred contextually from the current mark price and the state of the target account. These values could change before the liquidation is confirmed, particularly during periods of high Ethereum congestion. In many cases the liquidator would accept any successful liquidation, but for the sake of predictability, consider allowing the liquidator to specify an acceptable price range, the side of the positions to liquidate and a deadline for when the liquidation attempt expires.

Similarly, when creating the AMM pool, adding liquidity or removing liquidity, the price is inferred contextually. In the case of adding or removing liquidity, it could even change based on the actions of other users in the same block. Consider allowing the user to specify an acceptable price range and a deadline for when the action expires.

Update: *Acknowledged. The Monte Carlo team have decided not to address this issue.*

[M03] Re-entrancy possibilities

Solidity recommends the usage of the Check-Effects-Interaction Pattern to avoid potential security issues, such as reentrancy. However, there are several examples of interactions preceding effects:

- In the `deposit` function of the `Collateral` contract, collateral is retrieved before the user balance is updated and an event is emitted.
- In the `_withdraw` function of the `Collateral` contract, collateral is sent before the event is emitted
- The same pattern occurs in the `depositToInsuranceFund`, `depositEtherToInsuranceFund` and `withdrawFromInsuranceFund` functions of the `Perpetual` contract.

It should be noted that even when a correctly implemented ERC20 contract is used for collateral, incoming and outgoing transfers could execute arbitrary code if the contract is also ERC777 compliant. These re-entrancy opportunities are unlikely to corrupt the internal state of the system, but they would effect the order and contents of emitted events, which could confuse external clients about the state of the system. Consider always following the “Check-Effects-Interactions” pattern.

Most of the contracts and functions in the audited code base lack documentation. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. 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).

Additionally, the OpenZeppelin team found a notable lack of comments throughout the audited code. Well-commented code not only improves audit speed and depth, but it also helps to reveal what developer intentions may be and thus helps identify issues of misalignment between intention and implementation. Without comments, identifying issues in the code is much more difficult. Aside from benefiting auditors, code comments also benefit future developers and users, by clearly defining the functionality of the code and by reducing the risk of bugs.

Consider thoroughly commenting the existing code, and adding regular commenting to the software development process. We specifically recommend commenting every line of assembly code and commenting all complex math operations.

Update: *Fixed. Many parts of the codebase have been documented and commented.*

[M05] Incomplete Denial of Service prevention

As described in the documentation, the purpose of the broker and withdraw time locks is to prevent traders nullifying trades that have already been submitted. If this were possible, it would not only disrupt their counterparty's trade, it may also revert unrelated trades if they are processed in batches. The time locks ensure the order book has time to respond to a trader that attempts to withdraw excess collateral or make trades through the AMM or another broker. However, we have identified two other scenarios that would allow the trader to nullify their trade.

time lock, so a large `appliedBalance` can be used immediately to nullify the trade. Consider updating `availableMarginWithPrice` to check the trader's `appliedHeight` before processing the `appliedBalance`.

Secondly, a trader can directly cancel their order. This should immediately prevent it being processed, which would nullify the trade directly. Consider preventing traders from cancelling their request directly, and instead rely on their broker to relay the cancellation.

Note: This issue is related to C02 and any mitigation should consider both issues simultaneously.

Update: *Fixed with notable change to functionality. The entire timelock mechanism for brokers and withdrawals has been removed. This means there is no protection against a user nullifying a trade that was submitted to the offline order book. Importantly, this does not lead to inconsistencies within the protocol. Additionally, users can no longer choose an arbitrary broker address – they must use an address approved by the administrator (or the AMM).*

[M06] Governance parameter changes should not be instant

Many sensitive changes can be made by any account with the `WhitelistAdmin` role via the functions `setGovernanceParameter` within the `AMMGovernance` and `PerpetualGovernance` contracts. For example, the `WhitelistAdmin` can change the fee schedule, the initial and maintenance margin rates, or the lot size parameters, and these new parameters instantly take effect in the protocol with important effects.

For example, raising the maintenance margin rate could cause `isSafe` to return `False` when it would have previously returned `True`. This would allow the user's position to be liquidated. By changing `tradingLotSize`, trades may revert when being matched, where they would not have before the change. These are only examples; the complexity of the protocol, combined with unpredictable market conditions and user actions means that many other negative effects likely exist as well.

Since these changes are occasionally needed, but can create risk for the users of the protocol, consider implementing a time-lock mechanism for such changes to take place. By having a delay

can follow the same pattern as applying for withdrawals (adding an `applyAfter` timestamp, and checking it when the withdrawal is executed). If a timelock is implemented for governance parameter changes, the delay should be on the order of multiple days, to give users time to consider the effects of potential changes and act accordingly.

Update: *Acknowledged. The Monte Carlo team will implement a delay in the external governance mechanism.*

[M07] Undocumented Solidity assembly block

The `hashOrder` function uses assembly to hash an `Order` object. Although implemented correctly, the use of assembly discards several important safety features of Solidity, which may render the code unsafe or more error-prone. It is also harder for readers to parse, for reviewers to verify, and for developers to update.

Consider using `abi.encode` to achieve the same functionality. Alternatively, if the efficiency of assembly is required, consider commenting each line, including the magic constants. Additionally, the current pattern accesses unrelated memory locations and introduces low-level assumptions, such as the struct not being allocated below address 32. These should be thoroughly documented.

Update: *Fixed. The assembly block has been commented.*

[M08] `wfrac` function returns unintended value in edge case

The `wfrac` function in the `LibMathSigned` library is intended to return `x * y / z`. However, in the case of `(x, y, z) = (1, _INT256_MIN, -1)`, the intended result will not be returned. This is because `_INT256_MIN` has a higher absolute value than the maximum representable `int256` value, due to the way numbers are encoded in two's complement format.

When `t` equals `_INT256_MIN` and `z` is less than `0`, the negation of `t` will return the same value as `t`, `_INT256_MIN`.

To avoid this issue, consider replacing the implicit negation of `t` with the call `t.neg()`. The `neg` function will revert when called with `_INT256_MIN`. Consider also applying this to the implicit negation of `z`.

As discussed in issues M01 and M06, `tradingLotSize` and `lotSize` depend on each other, and cannot be changed such that `tradingLotSize % lotSize != 0`.

For these two dependent parameters, consider creating some function that changes both atomically.

Update: *Acknowledged.*

Low severity

[L01] Position size update

Whenever the `UpdatePositionAccount` event is emitted, the `perpetualTotalSize` parameter is set to the size of the `LONG` position, regardless of which side was updated. Although the two sides should eventually balance, the event is often emitted while they are still unbalanced:

- when it is emitted the first time during a trade, the second position has not been updated, so the two sides will differ by the trade amount.
- when it is emitted the first time during a liquidation, the liquidator's position has not been updated, so the two sides will differ by the liquidation amount.
- when it is emitted during settlement, the positions are closed in an arbitrary order, so the two sides may have different sizes.

Consider emitting the total size of the relevant position, instead of defaulting to the `LONG` size, which is often incorrect.

Update: *Partially fixed. The `liquidate` function no longer emits an `UpdatePositionAccount` event. The remaining events now emit the total size of the relevant position.*

[L02] Fragile signature verification

ECDSA signatures are inherently malleable, where it is possible to modify a valid signature to produce an equivalent signature over the same data. In Ethereum, this has been addressed for

Additionally, `ecrecover` returns zero when the signature fails, but this condition is not validated in `LibSignature`. Therefore, it is possible to produce a valid order on behalf of the zero address.

Although neither of these edge cases can be exploited directly in the current system, it introduces unnecessary fragility. Consider importing and using the `recover` function from OpenZeppelin's ECDSA library not only to benefit from bug fixes to be applied in future releases, but also to reduce the code's attack surface.

Update: *Fixed. A `recover` function is added in the `LibSignature` library to remove the reported fragilities by adding checks on `s` and `v`. This function also checks whether the returned value from `ecrecover` is zero. Please note that these checks are based on the OpenZeppelin's ECDSA library `recover` function that is imported in the code but never used.*

[L03] Overwriting cash balance

The `setCashBalance` function can be used to manually adjust a trader's cash balance during the `SETTLING` phase. However, if a separate balance-changing transaction (such as `deposit` or `liquidate`) is confirmed first, the adjustment will not account for the new balance. Consider replacing the `setCashBalance` function with `increaseCashBalance` and `decreaseCashBalance` functions to avoid the possible race condition.

Update: *Fixed. The `setCashBalance` function has been replaced by `increaseCashBalance` and `decreaseCashBalance` functions.*

[L04] Erroneous docstrings and comments

Several docstrings and inline comments throughout the code base were found to be erroneous and should be fixed. In particular:

- The NatSpec comment on line 20 of `LibSignature.sol` states that `hash` is the result of an EIP-712 hash, when it can actually be this or a normal Ethereum hash on a bytes structure.
- On line 111 of `AMM.sol`, the comment should say `last* functions are calculated based on on-chain fundingState`.

Update: *Partially fixed. The NatSpec comment in* `LibSignature.sol` *and the repeated comment in* `AMM.sol` *have been fixed. However the docstring above the* `current*` *functions in the* `AMM.sol` *still mentions that* `current* functions are calculated based on the on-chain fundingState` *instead of* `last* functions are calculated based on the on-chain fundingState`

[L05] Commented out code

The functions `isEmergency` and `isGlobalSettled` in the `IPerpetualProxy` contract are commented out without giving developers enough context on why those lines have been discarded, thus providing them with little to no value at all.

Similarly, the functions `roundFloor` and `roundCeil` in the `LibMathSigned` library are commented out.

To improve the readability of the code, consider removing the commented lines from the codebase.

Update: *Fixed. Commented code has been removed from* `LibMathSigned`*.* `IPerpetualProxy` *contract has been removed altogether.*

[L06] Incorrect guard condition

The guard condition that limits withdrawals from the insurance fund compares the raw token value to the scaled WAD balance. Therefore, it doesn't actually protect against withdrawing too much. In practice, the subsequent sanity check will catch any errors. Nevertheless, for clarity, consider updating the guard condition to compare values of the same scale.

Update: *Fixed. The proper check has been implemented.*

[L07] Transfer of ETH may unexpectedly fail

In the `_withdraw` and `withdrawFromProtocol` functions of the `Collateral` contract, the transfer of Ether is executed with Solidity's `transfer` function, which forwards a limited amount of gas to the receiver. Should the receiver be a contract with a fallback function that needs more than 2300 units of gas to execute, the transfer of Ether would inevitably fail. After the Istanbul

To avoid unexpected failures in the withdrawal of Ether, consider replacing `transfer` with the `sendValue` function available in the OpenZeppelin Contracts library.

Update: *Fixed. The `transfer` function has been replaced with the `sendValue` function.*

[L08] Use of arithmetic operators instead of `LibMath` functions

Given below is the list of occurrences in the codebase where arithmetic calculations are performed using arithmetic operators instead of using `LibMath` library functions:

- In `ChainlinkAdapter.sol` : For calculating the `newPrice`, multiplication is done using the `*` operator rather than using the `LibMathSigned.mul` function
- In `InversedChainlinkAdapter.sol` : For calculating the `newPrice`, multiplication is done using the `*` operator rather than using the `LibMathSigned.mul` function
- In `Collateral.sol` : implements subtraction of `decimals` from `MAX_DECIMALS` using the `-` operator instead of `LibMathUnsigned.sub` function
- In `AMM.sol` : multiplication between two integers are performed using `*` operator

To safeguard from code overflows, underflows and arithmetic errors, please consider using `LibMath` libraries. Also note that the `LibMath` libraries are using `*` and `/` operators instead of relying on their own safe methods. Examples can be found here and here

Update: *Fixed. Suggested changes have been made.*

[L09] Hard-coded order version

In the `Exchange` contract, the constant SUPPORTED_ORDER_VERSION is set to the value `2`, but is not used in the code base. However, the function `validateOrderParam` ensures that the order's version is `2`.

Since it appears that these two parts of the code are related, consider using `SUPPORTED_ORDER_VERSION` in the guard condition. This will ensure that changing this value will impart the expected behavior, and will improve code readability and auditability.

[L10] Implicit input value assumption

In the `LibMath` library the `wpowi` function is assuming implicitly that `n` is non-negative. It actually returns the wrong answer if a negative `n` is used.

In order to avoid wrong returned values, consider validating the input within `wpowi` before executing the calculations.

Update: *Fixed. The `wpowi` function now specifically accepts only non-negative values of `n`.*

[L11] ShareToken contract details

The `decimals`, `name` and `symbol` variables of the `ShareToken` contract are explicitly defined. While OpenZeppelin Contracts are already used, consider using the `ERC20Detailed` contract, which contains the additional variables.

This would change `decimals` to a `uint8` instead of a `uint256`, making it consistent with the standard.

Note that in the newest package version these details of an `ERC20` contract have been included in the main contract.

Update: *Fixed. `ShareToken` now inherits `ERC20Detailed`.*

[L12] Unclear variable names

The variable name `guy` is used frequently throughout the code. To favor explicitness and readability, we suggest renaming all instances of the variable `guy` to `trader` in `Collateral.sol`, `Position.sol`, in `Perpetual.sol`, `Exchange.sol`, `ContractReader.sol`, `PerpetualProxy.sol` and `AMM.sol`, as well as any corresponding occurrences within interface contracts.

Additionally, in the `LibMath` library the `_UINT256_MAX` variable is declared as `uint256` with value `2**255 -1`. This is not actually the maximum value for `uint256` variables, which is `2**256-1`. This assignment makes sense since the constant is the maximum value that can be represented by an `int256`. Nevertheless, in order to avoid confusion for future developers,

Update: *Fixed. The variable* `guy` *has been renamed to* `trader` *throughout the code base and* `_UINT256_MAX` *has been renamed to* `_POSITIVE_INT256_MAX`*.*

Notes & Additional Information

[N01] WhitelistedRole and ERC20Mintable removed from OpenZeppelin Contracts v3.0.0

The codebase utilizes `WhitelistedRole.sol` from OpenZeppelin Contracts. For example, it is imported here in `GlobalConfig.sol`. This contract was supported up to OpenZeppelin Contracts version 2.5.0, but beginning with version 3.0.0, it has been replaced with `AccessControl.sol`. Similarly, `ERC20Mintable.sol`, which is imported into the codebase here, no longer exists in OpenZeppelin Contracts v3.0.0.

Although the team maintaining OpenZeppelin Contracts will continue to provide support for security issues, new functionality will no longer be added to the `WhitelistedRole` nor the `ERC20Mintable` contracts. `AccessControl.sol` was designed with consideration for common use cases, security, and complex systems. To keep up with constantly-improving OpenZeppelin Contracts and future Solidity developments, consider updating the codebase to utilize the latest version of OpenZeppelin Contracts.

Update: *Acknowledged by the Monte Carlo team.*

[N02] Lack of explicit visibility

The `perpetual` state variable in the `PerpetualProxy` contract implicitly uses the default visibility.
Moreover the `IPerpetualProxy` interface declares a `perpetual` function that matches the one that is silently implemented in `PerpetualProxy` as the automatically generated getter.

To favor readability, consider explicitly declaring the visibility of all state variables and constants.

Update: *This is not an issue anymore since the* `perpetual` *state variable has been removed along with the* `IPerpetualProxy` *contract.*

necessitates custom getter functions to extract the individual components. To favor simplicity and readability, consider using a data type with named variables to address the individual fields.

It should be noted that the data type can be densely packed, so the underlying storage layout would not be affected. Furthermore, the object could be cast to the existing `Order` struct before generating the EIP712 hash, so compliance with the standard wouldn't require any additional complexity to handle the extra fields.

Update: *Acknowledged by the Monte Carlo team. The `Order` struct's `data` field remains unchanged, it is designed to make the `Order` data structure compatible with old off-chain infrastructure.*

[N04] Unused `OrderSignature` struct

The `LibEIP712` library declares an unused `OrderSignature` struct that is a duplicate of the one declared in the `LibSignature` library. To avoid confusion and inconsistencies, consider removing it.

Update: *Fixed. The unused `OrderSignature` struct has been removed from `LibEIP712` library.*

[N05] Unused import statements

Consider removing the following unused imports:

- `LibOrder` in `LibTypes.sol`
- `LibMathUnsigned` in `ChainlinkAdapter.sol`
- `LibMathUnsigned` in `InversedChainlinkAdapter.sol`
- `LibOrder` in `AMM.sol`
- `IPriceFeeder` and `IGlobalConfig` in `Perpetual.sol`
- `IERC20` and `SafeERC20` in `Perpetual.sol`, once the unnecessary association is removed.
- `LibOrder` in `Perpetual.sol`, once the unnecessary association is removed.
- `LibMathSigned` and `LibMathUnsigned` in `PerpetualProxy.sol`
- `ERC20` in `ShareToken.sol`

- `LibOrder` in `LibTypes.sol`

Update: *Partially fixed. All of the unused imports mentioned above are removed except* `LibOrder` *which is still imported in* `Perpetual.sol` *and is used in this unnecessary association.*

[N06] Unused constant

In the `AMM.sol` contract, the constant `ONE_WAD_U` is declared but never used. Please consider removing it.

Update: *Fixed. The unused constant* `ONE_WAD_U` *has been removed from the* `AMM.sol` *contract.*

[N07] Unused return parameters

In the `Perpetual` contract, the `liquidateFrom` function is declared as returning two `uint256` unnamed variables that are never returned or explicitly set inside the function.

Consider removing it from the definition to reflect the actual behavior of the function.

Update: *Fixed. The function* `liquidateFrom` *has been refactored to the* `liquidate` *function which now returns two* `uint256` *variables* `liquidationPrice` *and* `liquidationAmount`.

[N08] Unused `OrderStatus` enum

Please consider removing the unused enum `OrderStatus` declared in `Exchange.sol`

Update: *Fixed. The unused enum* `OrderStatus` *has been removed from* `Exchange.sol`.

[N09] Uninformative error messages

Several error messages in require statements were found to be too generic, not accurately notifying users of the actual failing condition causing the transaction to revert. Within the following files, we identify spots where error messages could be changed for greater clarity.

- on line 225, consider using the message “amount must be divisible by tradingLotSize”
- on line 259, consider using the message “amount must be divisible by lotSize”
- on line 287, consider using the message “shareBalance too low”
- on line 622, consider using the message “time steps (n) must be positive”

- within `Brokerage.sol`:

- on line 25, consider using the message “invalid newBroker”

- within `Collateral.sol`:

- on line 123, consider using the message “negative balance”

- within `Exchange.sol`:

- on line 63, consider using the message “amounts must be divisible by tradingLotSize”
- on line 74, consider using the message “taker initial margin unsafe”
- on line 103, consider using the message “maker initial margin unsafe”
- on line 119, consider using the message “amount must be divisible by tradingLotSize”
- on line 206, consider using the message “available margin too low for fee”

- within `LibMath.sol`:

- on line 99, consider using the message “cannot convert negative ints”
- on line 294, consider using the message “cannot convert uints greater than `_UINT256_MAX`“

- within `Perpetual.sol`:

- on line 53, consider using the message “fallback function disabled”
- on line 69, consider using the message “cannot deposit to 0 address”
- on line 157, consider using the message “cannot withdraw from 0 address”
- on line 168, consider using the message “withdrawer available margin negative”
- on line 194, consider using the message “no ether sent”

Error messages are intended to notify users about failing conditions, and should provide enough information so that the appropriate corrections needed to interact with the system can be applied. Uninformative error messages greatly damage the overall user experience, thus lowering the system's quality. Therefore, consider not only fixing the specific issues mentioned, but also reviewing the entire codebase to make sure every error message is informative and user-friendly enough.

Update: *Partially fixed. Most of the listed error messages are more descriptive or no longer necessary.*

[N10] Inconsistent use of return variables

There is an inconsistent use of named return variables across the entire code base. For example, here the returned variables are named but not assigned to, here they are unnamed and here they are never returned.
Consider removing all named return variables, explicitly declaring them as local variables where needed, and adding the necessary return statements where appropriate. This should improve both explicitness and readability of the project.

Update: *Partially fixed. While the mentioned examples have been rectified, there are still inconsistencies in the codebase. For example, the* `wfrac` *and* `wpowi` *functions in* `LibMathSigned` *still use named return variables.*

[N11] Liquidation price is not validated

The `calculateLiquidateAmount` function doesn't confirm that `liquidationPrice` is not zero. In this scenario, the function would revert when performing a division by zero. Following the "fail early and loudly principle", consider explicitly requiring the price to be positive.

Update: *Fixed in PR#5.*

[N12] Redundant condition in `require`

On line 30 of Collateral.sol, a `require` is done which checks the conditions: `_collateral != address(0x0) || (_collateral == address(0x0) &amp;&amp; decimals`

`18` will produce the exact same result.

Consider simplifying the conditional statement to make the code smaller and more readable.

Update: *Fixed. The `require` statement has been updated.*

[N13] Misplaced state variable definitions

In the `LibMath` library there are some state variables declared in between two functions.

To improve readability and follow the Solidity style guide, consider positioning the state variables before the constructor.

Update: *Fixed. The state variables declaration is positioned as per the suggestion.*

[N14] Inconsistency between “emergency” and “settling”

In the `ContractReader` contract the `isEmergency` parameter of the `PerpetualStorage` struct is `true` only if the perpetual contract is in `SETTLING` mode.

Moreover the documentation clearly states that when settling, the system enters into an `Emergency status` where trades and withdrawals are disabled until the settlement ends.

Consider unifying the variable names in the code and documentation to improve the readability of the code and avoid confusion.

Update: *Fixed. All occurrences of `SETTLING` have been renamed to `EMERGENCY` in the code base. Note that the process is still described as “settlement” and the final state is `SETTLED`.*

[N15] Inconsistency in defining 10**18

Throughout the codebase, the use of the value `10**18` has been significant. However, there is an inconsistency in defining this value.
For example, in the `LibMath` library the `_WAD` variable is declared along with its getter as `10**18` within both `LibMathSigned` and `LibMathUnsigned`. However, `LibOrder`, which makes use of `LibMath`, declares the variable `ONE` as `1e18`. Also, `AmmGovernance`

Other occurrences of this value in the code are:

- constant `ONE_WAD_U` of `AMM`
- constant `ONE_WAD_S` of `AMM`
- The `PerpetualGovernance` initial margin rate restriction
- constant `ONE` of `InversedChainlinkAdapter`

Given the role `10**18` has inside the system, consider defining this variable only once (or twice, if a signed and unsigned version are desired) and refactor any use to use this definition.

Update: *Partially fixed.* `ChainlinkAdapter` *still uses a new constant* `ONE` *instead of the* `WAD()` *function of* `LibMathUnsigned` *library.* `AmmGovernance` *and* `PerpetualGovernance` *still use* `10**18` *directly.*

[N16] Inconsistency in defining 'other' large constants

Apart from `10**18`, there are several large constants defined inconsistently throughout the codebase.
For example, `10**x` syntax is used in `chainlinkDecimalsAdapter` of `ChainlinkAdapter` whereas raw decimal syntax is used in `fixed_1` of `LibMathSigned`

Other occurrences of large decimals in the code are :

- `chainlinkDecimalsAdapter` of `InversedChainlinkAdapter`
- `longer_fixed_1` of `LibMathSigned`
- the `LibMathSigned` natural log restriction
- `FEE_BASE_RATE` of `LibOrder`

To improve the readability of the code, please consider using `10**x` or `1ex` syntax for all the large constants that can be expressed in this form.

Update: *Partially fixed. All of the examples mentioned above have been fixed except the* `LibMathSigned` *natural log restriction.*

private function `_withdraw` has an underscore `_` prepended while the private function `withdrawFromAccount` from the `Perpetual` contract doesn't.

Taking into consideration how much value a consistent coding style adds to the project's readability, enforcing a standard coding style with help of linter tools such as Solhint is recommended.

Update: *Fixed.*

[N18] Lot size checks are generalizeable

Throughout the code, there are many instances of checking that some amount, after running a modulo of `tradingLotSize` on it, returns `0`. One such check exists on line 63 of `Exchange.sol`, while another exists on line 119. Similar checks involving `lotSize` exist within `AMM.sol` here and here.

Since this code is repeated many times throughout the codebase, consider implementing it within a function or modifier, and instead using this whenever needed. By doing so, the code will follow the "don't repeat yourself" development principle, and the surface for error will be reduced since all instances of this code will behave the same way.

Update: *Acknowledged.*

[N19] Repeated code

- In the `Collateral` contract, the _withdraw function in lines 81-85 is doing the same operations as in the function `withdrawFromProtocol`

To improve readability and reduce code size, consider encapsulating the repeated code into a single function that can be called when needed.

Update: *Fixed. The repeated code has been encapsulated into the `pushCollateral` function.*

[N20] SafeMath and SignedSafeMath should be used

`SignedSafeMath` library. Likewise, the functions `mul`, `div`, `sub`, `add`, and `mod` within `LibMathUnsigned` match the functionality of OpenZeppelin's `SafeMath` library.

Consider removing the duplicated functions from the `LibMathSigned` and `LibMathUnsigned` libraries, and instead using the OpenZeppelin libraries as-is, within their own files. Everywhere that imports `LibMathSigned` or `LibMathUnsigned` can import `SignedSafeMath` or `SafeMath`, with a corresponding `using (library) for (type)` statement. By using `SafeMath` and `SignedSafeMath` within their own files, updates to the libraries will be much easier to apply. Additionally, the potential for error from re-implementing `SafeMath` and `SignedSafeMath` functions will be completely eliminated.

Update: *Acknowledged. The Monte Carlo team intends to address this when upgrading to Solidity 0.6.x.*

[N21] Typographical errors

- In line 419 of `AMM.sol`: "already" is misspelled
- In line 37 of `LibTypes.sol`: "current" is misspelled
- In line 42 of `LibTypes.sol`: "applied" is misspelled
- In line 79 of `Position.sol`: "negative" is misspelled
- In line 207 of `Position.sol`: "invalid" is misspelled
- In line 265 of `Position.sol`: "liquidated" is misspelled
- In line 279 of `Position.sol`: "position" is misspelled
- In line 292 of `Position.sol`: "negative" is misspelled
- In line 68 of `Collateral.sol`: "negative" is misspelled
- In line 7 of `GlobalConfig.sol`: "submiting withdrawal apllication" should be "submitting application"

Update: *Partially fixed. Some of the examples mentioned above have been corrected or removed. However, there are some typographical errors in `MarginAccount`, which can be found here, here and here.*

[N22] Inherit from interfaces

Consider updating the inheritance declarations of the `AMM`, `GlobalConfig`, `Perpetual` and `PerpetualProxy` contracts to implement the relevant interfaces.

Update: *Acknowledged.*

[N23] Missing view modifier

For added readability and better code readability, consider declaring the following functions as `view` since all of them preserve the contract's storage:

In `Position` contract:

- `availableMarginWithPrice`
- `marginBalanceWithPrice`
- `calculatePnl`
- `pnlWithPrice`

In `Perpetual` contract

- `isIMSafeWithPrice`
- `isSafeWithPrice`

In `PerpetualProxy` contract:

- `isProxySafeWithPrice`
- `isSafeWithPrice`
- `isIMSafeWithPrice`

Update: *Not an issue. These calculations may update the funding.*

Conclusion

2 critical and 1 high severity issues were found. Some changes were proposed to follow best practices and reduce potential attack surface. We later reviewed all fixes applied by the Monte Carlo team and **all most relevant issues have been fixed**.

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