PUBLIC

Code Assessment

of the Boros Markets Smart Contracts

08 August, 2025

Produced for

by

Contents

1 Executive Summary

Dear Pendle team,

Thank you for trusting us to help Pendle with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Boros Markets according to Scope to support you in forming an opinion on their security risks.

Pendle implements Pendle Boros, a marketplace for Interest Rate Swaps based on oracle-reported rates and an on-chain orderbook, allowing cross-margined markets and leverage.

The most critical areas addressed in our audit are asset solvency, resistance to manipulation, and the precision of arithmetic operations. Security regarding asset solvency is high. In the first version (out of 5 versions), an empty orderbook could lead to bad debt due to filling of orders at extreme prices, see High Priced Buy Order on Empty Orderbook Can Generate Bad Debt. The margin system has been revamped since version 2, fully resolving the issue by restricting the range in which orders can be created and enabling the admin to purge orders that fall outside this range. Resistance to price manipulation is good after the maximum allowable changes in the TWAP price have been lowered. TWAP instability could also be a concern if the spread is high, though it can be mitigated by Pendle's intention of lowering the spread and increasing the TWAP duration. Security regarding arithmetic operations is high. The rounding is performed in favor of the system and calculations are done with high precision.

General topics covered include code complexity, documentation and decentralization. Security regarding code complexity is good. The codebase is well-structured, though it makes extensive use of inline assembly, which bypasses many built-in safety checks. Security regarding documentation is high, with both a whitepaper and a specification available. Decentralization is improvable. Risk operations that are required to maintain the economic security of the protocol, such as order cancellation, order purging, and liquidations, are permissioned and can only be performed by whitelisted accounts. Users must trust the admin to perform these obligations at all times for the protocol to remain solvent.

In summary, we find that the codebase provides a good level of security. However, the settlement process, involving `FTags`, `TickNonceData`, `MatchEvent`, Quaternary Indexed Trees, and optimized sorting (`LibOrderIdSort`), is exceptionally complex. While designed for efficiency, such complexity significantly increases the surface area for subtle bugs related to state consistency, off-by-one errors, or incorrect handling of edge cases. The margin logic is also mathematically complex, and incorrect mathematical modeling of the system, which is out of scope of this review, might lead to insolvency of certain users. The risks are mitigated by the upgradeable and pausable nature of the contracts. We recommend that Pendle implements extensive monitoring of the protocol to swiftly react in case of anomalies. The security of the system also vitally depends on the correct selection of market parameters, such as TWAP time window, maximum rate deviation, margin factors, and more. It is the responsability of Pendle to choose parameters that ensure the security of the system.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	1
• Code Corrected	1
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	10
• Code Corrected	7
• Specification Changed	1
• Risk Accepted	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Boros Markets repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	9 March 2025	d479e388725aeae4ce5334dc34e480d9b6b92681	Initial Version
2	29 May 2025	4f769483ded7dff9afd226bba9e0e171c8e95a26	Second version
3	12 June 2025	8f9855c22ae4a1c733de8756e7ab26f6c7c7b7d3	Third version
4	20 July 2025	e2aa73d0c4b8d427bed661ee0a747b1d8e11aa6c	Fourth version
5	25 July 2025	a905b3788f13edba3bbdb7e49b5594ae51f28b31	Final version

For the solidity smart contracts, the compiler version `0.8.28` was chosen.

This review covers the smart contracts defined in the following files:

```
./contracts/core/market/MarketEntry.sol
./contracts/core/market/core/CoreOrderUtils.sol
./contracts/core/market/orderbook/OrderBookUtils.sol
./contracts/core/market/settle/PendingOIUtils.sol
./contracts/core/market/core/MarketInfoAndState.sol
./contracts/core/market/core/CoreStateUtils.sol
./contracts/core/market/margin/LiquidationViewUtils.sol
./contracts/core/market/margin/MarginViewUtils.sol
./contracts/core/market/settle/SweepProcessUtils.sol
./contracts/core/market/settle/ProcessUtils.sol
./contracts/core/market/MarketOffView.sol
./contracts/core/market/MarketOrderAndOtc.sol
./contracts/core/market/MarketSetAndView.sol
./contracts/core/market/findexOracle/FIndexOracle.sol
./contracts/core/market/findexOracle/SampleFundingRateUpkeep.sol
./contracts/core/market/settle/LibOrderIdSort.sol
./contracts/core/market/orderbook/TickBitmap.sol
./contracts/core/market/orderbook/Tick.sol
./contracts/core/markethub/MarketHub.sol
./contracts/core/markethub/Storage.sol
./contracts/core/markethub/MarginManager.sol
./contracts/lib/Errors.sol
./contracts/lib/math/TickMath.sol
./contracts/lib/math/PMath.sol
./contracts/lib/FixedWindowObservationLib.sol
./contracts/lib/ArrayLib.sol
./contracts/lib/PaymentLib.sol
./contracts/types/Order.sol
./contracts/types/createCompute.sol
```

```
./contracts/types/Account.sol
./contracts/types/MarketTypes.sol
./contracts/types/RecentTradeRateLib.sol
./contracts/types/MarketImpliedRate.sol
./contracts/types/TransientOrderIdMapping.sol
./contracts/types/StoredOrderIdArr.sol
./contracts/types/Trade.sol
./contracts/factory/MarketFactory.sol
```

In Version 2 the following contracts were included in scope:

```
./contracts/core/market/MarketRiskManagement.sol
./contracts/core/market/core/RateUtils.sol
./contracts/core/markethub/MarketHubEntry.sol
./contracts/core/markethub/MarketHubRiskManagement.sol
```

The following contracts were removed from the scope:

```
./contracts/core/markethub/MarketHub.sol
./contracts/core/market/findexOracle/SampleFundingRateUpkeep.sol
./contracts/types/RecentTradeRateLib.sol
```

In Version 4 the following contracts were included in scope:

```
./contracts/core/markethub/MarketHubSetAndView.sol
```

2.1.1 Excluded from scope

The economic soundness of the system and the mathematical proof of the security of the specification are explicitly out of scope for this review. The assessment focused on reviewing that Boros Markets implements the financial formulas as specified in the Boros whitepaper.

Furthermore any contracts not explicitly listed in the scope, including third-party libraries, are excluded from this review.

2.2 System Overview

This system overview describes Version 4 of the contracts as defined in the Assessment Overview.

At the end of this report section, we have added subsections for each of the changes according to the versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Pendle offers Boros, a marketplace for interest rate swaps using an on-chain order book.

Interest rate swaps are financial derivatives where two parties agree to exchange a fixed-interest income stream against a variable-interest stream. Parties agree on a notional amount, a maturity date, and a reference rate: a variable interest rate that is the object of the swap. The fixed-rate payer knows up front what they owe; the variable-rate payer pays based on the accrued variable rate. In Boros, the fixed interest is paid up-front, and the variable leg is paid at specified intervals. Multiple markets can exist for different reference rates, maturities, and notional currencies like ETH or USD.

Boros offers:

1. A public on-chain order book so a user (order taker) can swap with many parties at once (order makers).
2. An OTC channel, so two parties can enter a swap at a chosen fixed rate and size.
3. An AMM that provides liquidity to swap without active participation from liquidity providers.

Boros handles solvency by ensuring both counterparties hold enough collateral to cover their floating-rate obligations, and to cover liquidations in case of potential insolvency. Boros also aggregates order-book swaps into a Time-Weighted Average Price (TWAP) to produce a "mark rate", which is used to price positions for margin checks, liquidations, deleverages and purging.

From a user's perspective, their counterparty is the whole system (regardless if the positions are created through OTC, AMM, or order book). The system is solvent because it is backed by users with opposite positions. A user can take a LONG exposure (buying variable, pay fixed up front) or SHORT exposure (selling variable - receive fixed up front). If rates go negative, shorts receive variable payments and longs pay.

For each market, a user has a signed "size" (notional) of long (if size positive) or short (if size negative). The user also has a cash balance that can be for a specific market (isolated account) or shared between markets with the same notional currency (cross-margined account). The value of a user's positions plus cash must always be non-negative, otherwise the user is insolvent and cannot fullfil their obligations. In general, the system requires the value of a user positions and cash to be above a certain threshold. This is the margin check that keeps the system solvent.

Payments are made to or from a user's balance on a fixed time period, the payment amount is determined by the accrual of the variable interest since last update, which is reported on-chain via an oracle. Every period a fee (proportional to absolute size) goes to the protocol.

2.2.1 MarketHub

The MarketHub contract keeps track of users. Users are identified by *MarketAccounts*. A *MarketAccount* is made of:

- the users EVM address
- subaccount number (every user has up to 256 subaccount)
- the *tokenId* that this MarketAccount uses: every notional currency has a specific ID inside Boros. A given *MarketAccount* can only interact with Markets for its specific currency.
- The *marketId* of the market that this accounts operates, on, or CROSS, if this account is a cross-margined account that operates on multiple markets simultaneously.

The *MarketHub* keeps track of the cash balance of accounts, and the markets that they have entered. An account can enter a single market, if it is *isolated*, or multiple markets, if it is *CROSS*. Every Market represents a different interest rate swap type: different markets have different reference rates, maturity dates, parameters. But every market entered by a given account shares the same notional currency (*tokenId*).

It is the role of *MarketHub* to also check that accounts fulfill the margin requirements.

2.2.1.1 Margin Requirements

There are two types of margin requirements in Boros: the Initial Margin requirement, and the Maintenance Margin requirement.

Their fulfillment is formulated as satisfying the following inequality:

$$TOTAL\_VALUE >= TOTAL\_MARGIN$$

Where total value of an account is:

$$TOTAL\_VALUE := C + \sum_{i \in M} V_i$$

where C is the cash amount of the account, and is shared by all markets that the account has entered, and V_i is the value of the position of the account in market i.

The current value of a position is estimated (the future payout is still uncertain) as:

$$V_i = s_i r_i T_i$$

, where s_i is the size, which can be positive or negative, r_i is the *mark rate* APR of the Market, which is reported by an external oracle or derived from the recent trades in the order book, and T_i is the time to maturity in years.

In **Version 2** of the protocol, the calculation of margin requirements has undergone significant changes. The *initial margin* represents the amount of collateral required whenever a user's position is increased or its direction changed. It is determined by aggregating the margin requirements for the user's current position and their open orders:

$$INITIAL\_MARGIN := max(MARGIN_{orders_{long}} \mp MARGIN_{size}, MARGIN_{orders_{short}} \mp MARGIN_{size})$$

The margin requirements of all long orders are summed and aded to the margin requirements of the current position if the position is long (or subtracted if it is short). Similarly, for short orders, the margin requirements are summed, and the margin requirements of the current position are added if it is short (or subtracted if it is long). The *initial margin* requirements are then the maximum of both. Hence, for a user with a long position and open long orders the formula becomes:

$$INITIAL\_MARGIN := \sum_{i \in M} \sum_{o_i \in O} K_{IM} |s_o| \hat{r_{i,o}} \hat{T_i} + K_{IM} |s_i| \hat{r_i} \hat{T_i}$$

Here K_{IM} is the initial margin factor, s_i is the current position size, and $\hat{r_i}$ is the margin index rate, which is the maximum of a minimum lower bound and the mark rate. This ensures that the margin does not become too small near zero interest rates: $\hat{r_i} = max(|r_i|, r_{min})$. Likewise, $\hat{T_i}$ is the time to maturity, which is lower-bounded by a minimum time-to-maturity to prevent margin requirements from becoming excessively low as the position approaches maturity.

For the open orders, the size of the order is denoted as s_o, and the rate is calculated as $\hat{r_{i,o}} = max(|r_{i,o}|, r_{min})$, where $r_{i,o}$ is the fixed interest rate of the order at its tick. The time to maturity for orders is also lower-bounded by a minimum time-to-maturity, which is denoted as $\hat{T_i}$.

The *maintenance margin* is the amount of collateral required to keep a position healthy and is used to determine whether a position can be liquidated. The `MAINTENANCE_MARGIN` is calculated as follows:

$$MAINTENANCE\_MARGIN := \sum_{i \in M} K_{MM} |s_i| \hat{r_i} \hat{T_i}$$

Here K_{MM} is the maintenance margin factor ($K_{MM} < K_{IM}$), s_i is the position size in the market, $\hat{r_i}$ is the maximum of a minimum lower bound and the absolute mark rate and $\hat{T_i}$ is the lower bounded time to maturity.

In cases where $r_i \geq r_{min}$, $T_i \leq T_{min} K_{MM}$, and $s_i r_i \geq 0$, an alternative formula is used to ensure that positions are not liquidated due to the mark rate increasing close to maturity:

$$MAINTENANCE\_MARGIN := \sum_{i \in M} K_{MM} |s_i| (|r_i| T_i + \hat{r_i} (T_{min} K_{MM} - T_i))$$

The Initial Margin requirement must be satisfied whenever a user's position is modified. A *weak* margin check exists that allows reducing the absolute size of a user's position in one market even when they would not fullfill the initial margin check and without recalculating the margin across all positions. For this, the ratio of value to margin must not decrease with the position change. This enables a user to deleverage even if they no longer satisfy the initial margin check:

$$\Delta value \leq \Delta MM \cdot h_{crit}$$

Here h_{crit} is the critical health ratio, a risk parameter of the system that is below 100%.

If the Maintenance Margin requirement is at any moment violated, the user can be liquidated, and their risky position is sold at a discount to another user. If the total value of the violator is negative, it is too late, and the system is temporarily insolvent. The bad debt can be repaired by governance through the use of the `forceDeleverage()`, which allows to transfer a position to another to cover the bad debt.

Leveraged positions can be taken with values of K_i below 1. In general the margin factors are expected to be well below one, when the liquidity is sufficient for swift liquidations.

2.2.1.2 Liquidations

When the maintenance margin is no longer satisfied for a user, either because the mark rate has decreased or because cash has decreased to pay for periodical fees, the user becomes liquidatable. A liquidation can only be performed by an authorized address and consists of acquiring the user's size on a given market, priced at a value between the current mark rate. If the violator (the users being liquidated) is short, the liquidator receives the value of the position in cash, and takes over the position. If the violator is long, the liquidator pays for the long at current mark rate and receives the positive size. A liquidation incentive is added to the cash received to liquidate shorts, or subtracted from the cost paid to liquidate longs. The liquidation incentive is a fraction of the cost and is a function of the health of the violator, which is defined as `TOTAL_VALUE` over `TOTAL_MARGIN`. The incentive is always such that liquidations never worsen the health of a position, meaning that liquidation cannot create bad debt. If `TOTAL_VALUE` is already below 0, meaning health is negative, bad debt is already created and liquidations are no longer possible. The insolvency must be addressed by governance through the `forceDeleverage()` function.

Forced deleveraging is a function available to governance that allows transferring a position (size) between two accounts, at a rate between the current mark rate and the bankruptcy rate: `size` is subtracted from one account and added to the other, while `size * r * T` (cost) is added to one and subtracted to the other.

The MarketHub coordinates the margin checks, and holds the cash balance of accounts in its storage. However, the position sizes for each of the markets are held in the markets themselves, and accounts' positions in markets could have accrued interest payments that are not yet accounted in the MarketHub. The role of accounting the payments to be sent or received because of user positions is delegated to the markets themselves. A user needs to be *settled* in a market to synchronize its state with the MarketHub, such that margin requirements can be properly checked.

2.2.2 Markets

For every token, multiple markets can exist, each one with a given maturity and reference rate. Each market keeps track of user positions and each market maintains an orderbook exchange where orders are matched.

The state of a user in the market consists of its size, a list with its orders in the orderbook, that could be either open or filled, and the *FIndex* of the user. The *FIndex* is the mechanism that the Market uses to keep track of the payments that need to be sent or received from users because of the accrual of variable interest.

A *FIndex* is made of:

- The time at which the index update takes place (`FTime`)
- the new variable rate index (`floatingIndex`)
- the new fee index (`feeIndex`)

The market stores the FIndex of when each user has been last been brought up to date with the market. Markets have an update period, for example 8 hours, at which a new *FIndex* is published. A new *FIndex* allows computing the payment to be received or taken from each users for their position. The payment amount owed to/from a user is:

```
payment = (market.latestIndex.floatingIndex() - user.index.floatingIndex()) * user.signedSize / 10**18
```

A fee is also computed, and it is then transferred to Pendle treasury:

```
fee = (market.latestIndex.feeIndex() - user.index.feeIndex()) * user.signedSize.abs() / 10**18
```

As we said, the *FIndex* is updated periodically based on an external oracle. Users positions are updated based on the new *FIndex* lazily, only when requested through the MarketHub. A user position can skip any amount of *FIndex* updates before being synchronized, the end state being the same with frequent or infrequent synchronizations.

This view of the user update is however incomplete, as it does not account for existing orders of the user in the orderbook, that might have filled in the past but are not accounted in the user's size yet. The full process to synchronize a user is called *settling*

2.2.2.1 Settling an account

Several index updates can occur before the state of a user is synchronized between the *MarketHub* and the *Market*. Updates of the Market FIndex happen on a regular schedule, when variable interest updates are relayed on-chain after the *scheduled* update time has passed. When the *FIndex* is updated however users are not touched directly, they need to independently be settled to compute how much they need to pay, or they receive, from the latest variable rate accrual. Anybody can trigger the settlement, so in practice the changes of user balance following a rate update, and their effect on liquidations, are immediate.

The user size tells us how much variable interest the user should have received, between the last time they were settled and the new index. However, the user size can also have changed multiple times since the latest settlement of the user, because of filled orders. When orders are filled on the Order Book, they are not immediately added to the order maker's balance, because the system is designed to be able to fill many orders with a single match. The orders are marked as filled in a gas efficient way, and their time is also recorded, and they are added to the *size* balance of their creator only when the order maker is settled.

Settling a user happens in three steps:

1. Sweeping

The list of orders of a user, which is stored in the Market storage and contains open and filled orders, is traversed. Orders are added to this list when they are opened, but when they get filled, they are only marked as filled indirectly in the orderbook. So the list of orders is traversed, and the orders that have been filled are collected.

2. Processing

The filled orders are processed by finding at which `FTime` they were filled, then orders with the same FTimes are aggregated. The result is a list of sizes, costs at a set of sorted times. Cost is the annualized fixed interest received/payed for a given order.

For each FTime, the FIndex value at that time is retrieved, and the user's payments and fees are computed until that FTime. The annualized costs for the filled orders realized at that FTime are converted to cash payments, with a value that depends on the FTime. The change of size is applied to the user. After all the FTimes at which orders were filled have been processed, the user's index is the same as the Market index, and the total payments and fees of this settlement have been computed.

3. Synchronizing the MarketHub

The payments and the fees of the user are applied to the cash balance of the user in the *MarketHub*.

Step 1. uses an optimized algorithm to perform the sweeping in a sublinear amount of storage accesses, by exploiting the fact that orders are identified by *OrderIds*, which encode price and priority at a given price in it, such that they have an ordering such that if there are OrderIds `A` and `B`, with `A < B` and `B` has been filled, then `A` has also been filled. This allows sorting the orders by *OrderId*, and performing a binary

search to find the last order that has been filled. (in practice the algorithm is slightly more efficient, by doing a *partition search*).

2.2.2.2 OTCs and Orderbook

Positions in a Market can be opened *OTC* (Over the Counter): where two parties enter a swap with an arbitrary size (positive for one side and negative for the other), and cost (likewise).

Positions can otherwise be acquired through the Orderbook of the market:

Counterparties offer to sell or buy positions on the Orderbook, and order takers can match against those open orders. There are two sides: the short orderbook, with people selling variable interest, and the long orderbook (people buying variable interest). Orders have size and fixed rate, and are identified by an OrderId which includes the side (long or short), the fixed rate (tick), and the position of the order in that tick by priority (older orders are matched first).

Fixed rate is encoded as *ticks* corresponding to a half basis point increase on the previous tick, such that the APR for a given positive tick i is $1.00005^i - 1$. Negative ticks encode negative APRs: $-(1.00005^{-i} - 1)$.

On the order book, counterparties post maker orders to buy or sell floating at a given size and fixed rate (“tick”). Takers match against maker orders:

Orders from order makers are aggregated by ticks, with tick being the smallest price granularity. Order takers match against multiple orders at once in a tick, with older orders being matched first.

From the taker perspective, an order is sent to buy (or sell) a certain amount of size, at a limit fixed rate (tick) that the taker is willing to pay (or receive). A long taker order will be matched against the short orderbook, and viceversa for short taker order. For long orders, matches are searched starting at the lowest tick that has open orders, and then iterating through growing ticks until either the taker order is fully filled, the limit tick is reached, or the order book is emptied.

Within each tick: If taker size is bigger than the total tick size, the entire tick is consumed. Otherwise the tick is partially matched.

Partially matching a tick is an algorithmically optimized process, since potentially a big number of open orders are present in a tick. Traversing the list of open orders would be prohibitively expensive on-chain, since each order would have to be read from storage, which is expensive. Orders are therefore organized in a tree structure, called Fennwick-tree, that allows efficient computation of prefix-sums. Prefix-sums are pertinent because they allow to efficiently find the range of orders in a tick that fully matches the taker order. Fennwick trees, beside efficient prefix-sum computation, allow for efficient cancelling of existing orders.

The specific Fennwick tree used is a zero indexed quad-tree with 9 layers. Each tree contains as such up to 4**9 = 262144 orders. Orders are added sequentially to trees, and as a tree becomes full, a neighboring tree is created.

The specifics of the Fennwick tree allows matching any number of orders in a tree with at most 9 storage accesses.

When orders are filled, the current *FTime* (*FTag* in Version 2) is stored for retrieval in a dedicated structure in the tick, that allows storing the match times for any number of orders matched at once with a single storage write, and allows retrieving the match time of a specific order in a number of storage reads at most logarithmic to the number of periods in the market. This information is required when settling a user.

When an open order is only partially matched by a taker, extra care needs to be used since the order stays open, and will be filled completely only later: its fill time will not be tracked according to the usual mechanism, and its size is decreased in the orderbook. The fill time, size, and cost are tracked through a storage slot belonging to the Order maker, dedicated to partial fills information. If the slot is free, or it is occupied by a partial fill that happened during this same epoch, the partial fill can be added to the slot. The value in the slot will be, together with filled orders, accounted to the next time the maker is settled. However, if the slot is already occupied with a partial fill from a previous epoch, the partial maker needs

to be settled immediately, including this partial fill, since otherwise there would be no way to keep track of this partial fill. This means that matching an order on the order book can trigger the settlement of another user, the maker of the last partially filled order. Since every match partially fills at most 1 order, at most one other user is settled when matching against the order book, which is important for gas concerns, since settling users is gas intensive, and matching on the order book should be inexpensive.

2.2.2.3 Adding order on the orderbook

If the taker order is not completely filled, either because the limit tick is reached or because the order book is empty, the remaining order size is placed on the orderbook, where a new maker order is created, depending on the *Time in Force* of the user's order. The Time in Force is one of:

- GTC - will fill orders and add remaining order sizes to the orderbook.
- IOC - will fill orders and not add remaining order sizes to the orderbook.
- FOK - will fill orders and revert if not all orders are filled.
- ALO - will revert if orders get filled, else add orders to the orderbook.
- SOFT_ALO - will drop orders that could get filled and add remaining orders to the orderbook.

Users can have at most `maxOpenOrders` open at once. This limit is required to limit the possibly unbounded gas consumption of sweeping. Orders have some restrictions where they can be placed. In particular long orders can not be placed too high above the mark price and short orders too low below the mark price

2.2.2.4 Cancelling orders

Users can cancel their orders. A flag, `isStrictCancel` is joined to the cancel request, which consists of a list of orders to remove. If any order to be removed is not found (because it has been filled or already cancelled), and `isStrictCancel` is set, the cancelling reverts.

An authorized address can also `forceCancel()` user orders, meaning they are cancelled without their action, in case orders would be deemed risky.

An authorized address can also cancel multiple orders of a user whose health is below the critical health ratio.

An admin can purge orders that are significantly above the mark price (or below the mark price for short orders). This functions similarly to matching on the orderbook, except a special *FTag* is recorded in the match to signal that the orders are not filled but cancelled.

2.3 Trust Model

The master admin (i.e. `DEFAULT_ADMIN_ROLE`) is fully trusted. They can upgrade the MarketHub contract, register new markets and tokens or withdraw funds from the treasury. Furthermore they can pause the protocol and pause individual withdrawals. They are trusted to give special admin rights to a small set of addresses and revoke them in case the addresses get compromised.

Any user holding the `DIRECT_MARKET_HUB_ROLE` are fully trusted. They can operate on arbitrary accounts.

The `INITIALIZER_ROLE` is trusted to correctly initialize all contracts.

The `keeper` is fully trusted to update the market's *FIndex* accurately.

Any token that has other non-standard behavior like callbacks on receive, rebasing or has fees on transfer is not supported by the protocol.

Tokens with exceedingly low unit value, below $\$ 10^{-6}$ per token, are not supported. These tokens allow creating positions whose size overflows the `int128` type used for sizes.

The protocol is expected to get deployed to Arbitrum or Base chain. The L2 sequencer is trusted to process transactions in the order received.

2.4 Changelog

Version 4:

- Added new order type (Soft) Add Liquidity Only (`SOFT_ALO`).
- Allow only whitelisted accounts to liquidate or force cancel.

Version 2:

- Replaced the `DELEVERAGER_ROLE` by role-based admin permissions stored in an `PERM_CONTROLLER` and the `FINDEX_ORACLE_UPDATER_ROLE` by a `keeper` address. Added the option to have more fine-grained access-control for each admin function.
- Limit orders Initial Margin is now calculated using the tick price instead of the Mark Rate.
- Added alternate Maintenance Margin formula to use near maturity to prevent unfair liquidations when Mark Rate increases.
- Allow admin to deleverage users at the bankruptcy rate to cover bad debt.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	2

5.1 TWAP Instability Due to Bid-Ask Spread

Design Low Version 1 Risk Accepted

CS-BOROS-MKT-024

The mark price is calculated as the time-weighted average price (TWAP) of the most recent matched trades within a block, unless an external oracle is configured. In the orderbook, long and short orders cannot cross, so the highest bid and lowest ask are always separated by at least one tick (also called bid–ask spread).

This means that even small trades can influence the TWAP due to their direction alone. For example, if the last trade is a short, the TWAP will be slightly lower than if it was a long, since there is a spread between the best bid and best ask.

An adversary can exploit this by spamming the blockchain with tiny trades in the desired direction. For instance, to push the TWAP upwards, they can repeatedly submit small long trades that match against the short side of the orderbook. Even if a large short trade just matched with the long orderbook, enough small long trades will ensure the last matched rate is from the short orderbook.

If the spread is larger, the mark price can be moved by a greater amount. This creates an incentive for an adversary to manipulate the mark price and trigger liquidations of risky positions. For example, they can spam small trades in each block to move the mark price and then check in the next block if the price has moved enough to liquidate a position before anyone else can react.

Risk accepted:

Pendle answered:

> We understand the implication of price manipulation within the bid-ask spread related to TWAP instability. We accept this risk and will actively counter this problem by using internal bots (do minimal trade as well) as well as setting up smaller spread. The mark rate TWAP is also set to be significantly longer (5 min at the start), which largely mitigate the issue. As Boros grows in adoption

and when we decide to significantly shorten the twap, we will keep iterating on the definition of the Mark rate in future upgrades.

5.2 `replaceWindow()` Makes the Mark Rate Oracle Value Jump

Design | Low | Version 1 | Risk Accepted

CS-BOROS-MKT-012

The market administrator calling `replaceWindow()` causes the mark rate returned by `impliedRate.getCurrentRate()` to suddenly change. If the administrator sets a shorter window, the value will jump towards `lastTradedRate`, while if a longer window is set, the value will jump towards `prevOracleRate`. This potentially causes healthy positions to suddenly become liquidatable because of admin action and makes it profitable to back-run the call of `replaceWindow()` with liquidations.

This can be avoided by snapshotting the implied rate before the parameter change.

Risk accepted:

Pendle has accepted the risk, but has decided to keep the code unchanged. They expect the window size to only change in exceptional cases.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	1

- High Priced Buy Order on Empty Orderbook Can Generate Bad Debt Code Corrected

High-Severity Findings	1

- Weak Margin Check Allows Creating Bad Debt Code Corrected

Medium-Severity Findings	2

- TWAP Manipulation by Orderbook Clearing Code Corrected
- Unsafe Use of Narrow Type in Inline Assembly Code Corrected

Low-Severity Findings	8

- Force Cancel Risky Users Can Be Blocked Code Corrected
- Fee Can Make Liquidations Unprofitable Code Corrected
- Main Account Has Priority When Increasing Collateral Code Corrected
- Margin Rounding Allows Exiting With Open Positions Code Corrected
- No Minimum Position Size Requirement Code Corrected
- PREVRANDAO Opcode Behavior on L2 Chains Code Corrected
- forceDeleverage() Does Not Follow Specification Specification Changed
- signedSize Internal Method Returns Incorrect Result Code Corrected

Informational Findings	5

- Finalize Withdrawal Violates CEI Pattern Code Corrected
- Inconsistent Specification Code Corrected
- Misnamed Variables and Functions Code Corrected
- _getRateAtTick Does Not Function Over Its Whole Domain Specification Changed
- coverLength of Maximum Index Node Is Incorrect Code Corrected

6.1 High Priced Buy Order on Empty Orderbook Can Generate Bad Debt

Design Critical Version 1 Code Corrected

CS-BOROS-MKT-001

With an empty orderbook, highly priced buy orders can be created with little collateral such that when they are filled the order maker is immediately insolvent and the system incurs a loss.

The margin required to open a buy order of size S is $K_{IM}S\Delta_t r$, where K_{IM} is the margin factor, inverse of max initial leverage, Δ_t is the annualized time to maturity, r is the mark interest rate (APR). In a new

iteration of the code, the mark rate in the margin computation is substituted with f, the fixed rate of the order. When creating a long order, this margin is required such that the filling of the order will not bring the account above the maximum initial leverage. This does not work; in case we can create a long order at an arbitrary high price (empty orderbook on the short side).

Let us assume that attacker A place an order to buy size S at fixed rate f, and that current time to maturity is 1 (for simplicity). The attacker is required to cover the open order with an initial margin (**IM**) of $K_{IM}Sr$ (see below for version with margin rate given by fixed rate of the order). If the order is filled, the system **pays** to the counterparty Sf, and A **gets** a position with value Sr (assuming mark rate r is the market price).

For the market to be solvent, the margin plus the value A receives must be bigger than the value the counterparty receives:

$$IM + get \geq pay \Leftrightarrow K_{IM}Sr + Sr \geq Sf \Leftrightarrow Sr(1 + K_{IM}) \geq Sf$$

or equivalently f needs to satisfy $f \leq r(1 + K_{IM})$ in order for the amount paid out by the buy order to not exceed the value received plus the initial margin.

If the initial margin for the order is $K_{IM}Sf$, the requirement becomes

$$IM + get \geq pay \Leftrightarrow K_{IM}Sf + Sr \geq Sf$$

or equivalently $f \leq \frac{r}{1 - K_{IM}}$.

In both cases, if the f of the order (the fixed rate of the order) exceeds the that limit, the market will immediately be insolvent as the order is filled, at the profit of the counterparty. The attacker can self-fill from another account, such that they extract the profit.

For this attack to be possible it is required to place a LONG order at a high rate (or a SHORT order at a low negative rate). This would normally match against the existing SHORT (or LONG) orders in the order book. Clearing the orderbook beforehand is therefore required. The cost of clearing the orderbook can be however recovered by the attacker than can extract unlimited value from the market by repeating this strategy.

The following is a numerical example with concrete values:

```
current mark rate 10%
user A has 100 cash, no positions
IMFactor = 0.1
time to maturity = 1y

With empty order book, A places a LONG order of size 10 at 10'000% ->
IM = 10 (size) * 10'000% (fixed rate) * 1 (time to maturity) * 0.1 (IMFactor) =  100
cash >= IM, so they can place the order.

Order is immediately filled by user B:
B gets 10 * 10'000% = 1000 cash upfront.
The SHORT position they now hold is worth -10 (size) * 10% (markRate) = -1,
so they can withdraw most of the cash.

B withdraws ~1000, of which only 100 is backed by A.
A is immediately deeply unhealthy (cash = -900, position value = 1).
```

Code corrected:

The code went through 5 versions, and this issue is already fully resolved after the margin system was revamped in version 2:

Version 2 introduced price limits to prevent orders from being placed at arbitrarily high or low rates. For long orders, the price limits are defined as $r + C$ when the mark price is below the threshold $k_i Thresh$,

and $r*S$ when the mark price is above the threshold. The constant C and slope S are configurable parameters that can be changed by the admin.

```
function __calcRateBoundPositive(int256 rMark, uint256 k_iThresh, Side side) private view returns (int256) {
    if (rMark >= int256(k_iThresh)) {
        int16 slope = side == Side.LONG ? _ctx().loUpperSlopeBase1e4 : _ctx().loLowerSlopeBase1e4;
        return mulBase1e4(rMark, slope);
    } else {
        int16 constBase1e4 = side == Side.LONG ? _ctx().loUpperConstBase1e4 : _ctx().loLowerConstBase1e4;
        return addBase1e18And1e4(rMark, constBase1e4);
    }
}
```

Additionally, function `MarketRiskManagement.forcePurgeOobOrders()` has been added for an admin to purge all existing orders that have become out of the bounds as the mark price changed.

Similar limits have also been introduced for the lowest rate at which short orders can be placed.

Version 2 also introduced a withdrawal cooldown feature which prevents any attacker from immediately withdrawing from the system, rendering this attack or any other "in-a-block" attacks infeasible.

6.2 Weak Margin Check Allows Creating Bad Debt

Design High Version 1 Code Corrected

CS-BOROS-MKT-002

When a user changes their position in a market, the system checks if the new position is within the *initial margin limits*. For this, the `MarginManager` first performs a "weak check" that considers only the values of the position on this market. Only if the weak check fails, the system performs a "strong check," that takes the positions on all markets into account. To fulfill the "weak check", the position must fulfill two conditions. First, the margin requirements of the user must not increase as a result of their action. Second, the difference of the user's position value and cash, minus the initial margin requirements, must not decrease.

```
function _isEnoughIMWeak(
    VMResult preIM,
    int256 preCash,
    VMResult postIM,
    int256 postCash
) internal pure returns (bool) {
    (int256 postValue, uint256 postMargin) = postIM.unpack();
    (int256 preValue, uint256 preMargin) = preIM.unpack();

    if (postMargin > preMargin) return false;
    return (preValue + preCash - preMargin.Int()) <= (postValue + postCash - postMargin.Int());
}
```

It can be shown that if the user's position value and cash are above the initial margin requirements, before the user's action, the weak check ensures that the position's health increases as result of the action. However, if a user's position is below the initial margin requirements before, the check can be abused to create bad debt. Note that the initial requirements are stricter than the maintenance requirements. So, as long as the user stays above the maintenance margin requirements they cannot get liquidated.

A strategy of an attacker to exploit this behavior is to create a position at the initial margin and then wait for a mark price update that moves against the user. As their value decreases the sum of position value and cash decrease below the initial margin requirements. Denote D as the amount by which the user is below the initial margin requirements:

$$D = preValue + preCash - preMargin < 0$$

An attacker can then trade with another user to close their position. As seen in function `_isEnoughIMWeak`, if *postMargin* and *postValue* are zero, the check would a trade that ends up with a

negative cash amount *postCash* = *-D*. The negative cash amount can be considered as the bad debt to the system as the attacker has no incentive to pay it back. Note that the counterparty in this trade can also be controlled by the attacker, so the bad debt directly benefits the attacker. An OTC trade is the simplest way to execute this strategy, as the user can set arbitrary values for the trade. However, the attacker could alternatively make repeated "bad trades" with their controlled counterparty in the orderbook and slowly extract value this way.

As shown in the equation, the amount of value extracted depends on the size of *D*. It can be demonstrated that the maximum value of *D* is approximately proportional to the amount of collateral, the change in mark price and the margin factor. An attacker needs sufficient capital to extract a large amount of cash. For example, with a margin factor of 10% and a 5% price decrease, the attacker can create bad debt worth approximately 50% of their starting position.

Markets use a TWAP (Time-Weighted Average Price) for the mark price. Since a TWAP adjusts gradually over time, an attacker can exploit this by waiting for a sudden price drop. They can then create a position that falls below the margin requirements as the mark price slowly adjusts to reflect the new price.

Code corrected:

In Version 2, the margin system has been completely redesigned to fully resolve the issue. The function `MarginViewUtils._checkMargin()` implements a more restrictive version of the weak margin check, but uses the maintenance margin instead of the initial margin.

```
if (diffValue > diffMargin.mulFloor(critHR)) {
    return (true, true, _getIMAft(market, user));
}
```

The updated check ensures that, as long as the user is above the critical health ratio prior to the action, they remain above it afterward.

The user is not tested to be above the critical health ratio, except when an admin imposes it as a requirement in exceptional market conditions. However, since the critical health ratio is set below 1, the user would have needed to avoid liquidation for a prolonged period in order to fall below this threshold, which is unlikely in practice.

6.3 TWAP Manipulation by Orderbook Clearing

Design | Medium | Version 1 | Code Corrected

CS-BOROS-MKT-003

The Time-Weighted average price for implied rate can be significantly manipulated in a single block transition (timestamp increases by 1 second on Arbitrum, 2 seconds on Base). Arbitrum and Base run centralized sequencers that process transactions in FIFO orders. It is therefore possible to send two transactions, one after the other, and have a high probability of the transactions being included one after the other without any other in between. More importantly, the later transaction is included before the first one is published, such that the MEV generated by the first transaction can be extracted by the second transaction before other parties have time to react.

An attacker can exploit this fact to send two transactions that will be included in different blocks (because of skillful timing, or gas limits preventing the second transaction from making it in the same block), with no other transaction in-between. The first transaction can clear the order book and fill an order at a very high or very low rate. This will be reflected in the value of the implied rate in the next block. The attacker can then exploit the new mark rate of the next block to profit from liquidations caused by the mark rate change.

If the block time is two seconds, and the TWAP window is defined as 15 minutes, the new rate oracle value will be:

```
(prevOracleRate * (15*60 - 2) + lastTradedRate * 2) / (15*60)
```

or `99.77% * prevOracleRate + 0.22% * lastTradedRate`. The attacker controls `lastTradedRate`, and can set it as high as the highest rate allowed. If `k_tickSpacing` is 4, the maximum rate is around 70000%. `0.22% * 70000 == 155%`, meaning the oracle rate can be increased to more than 155% APR, which is surely enough to make all short positions liquidatable. Equivalently the attacker can do this for long positions by manipulating the rate downwards. The high cost of clearing the orderbook can be maybe be recovered, depending on the liquidity of the orderbook and the amount of positions made liquidatable

If `k_tickSpacing` is higher, it could be exploited even more effectively by an attacker, to completely drain the protocol: when tickspacing is 15, the maximum tick rate is 4e9 %, which means that even controlling 0.22% of the value of the oracle, an attacker can manipulate it arbitrarily high. If the attacker owns a long position, that long position will be hugely overvalued, allowing them to withdraw all the cash in the protocol.

Code corrected:

In Version 2, the risk of TWAP manipulation has been mitigated by introducing a restriction on the last trade rate. The last trade rate can no longer deviate by more than a certain percentage of the previous mark rate, as enforced by the following check:

```
function _checkRateDeviation(int256 lastMatchedRate, MarketMem memory market) internal view returns (bool) {
    return
        (market.rMark - lastMatchedRate).abs() <=
        mulBase1e4(market.k_iThresh.max(market.rMark.abs()), _ctx().maxRateDeviationFactorBase1e4);
}
```

This mechanism ensures that the mark rate (that is based on the last trade rate) cannot be manipulated beyond a certain threshold, hereby preventing the attack scenario described above.

6.4 Unsafe Use of Narrow Type in Inline Assembly

Correctness | Medium | Version 1 | Code Corrected

CS-BOROS-MKT-023

Internal function `wordSlot()` of `TickBitmap.sol` uses inline assembly to perform arithmetic operations on a `uint8` value without clearing the higher order bits which might be dirty, leading to incorrect results, as documented by solidity documentation.

Numerical types narrower than 256 bits can have dirty high-order bits, for example as the result of a narrowing cast, or as the result of reading from storage where other packed variables are present. The dirty bits are removed lazily by Solidity when an action is performed which requires clearing them: for example writing to memory, since the memory might be later used in a hash computation, or in arithmetic operations, since the dirty bits might influence the result of the operation. However, the dirty bits are not cleared automatically when a stack variable is used directly in inline assembly, therefore, when using types narrower than uint256 in assembly the effect of dirty bits should always be considered.

The following code in `TickBitmap.sol` accesses `wordPos` without clearing its high order bits:

```
function wordSlot(TickBitmap storage self, uint8 wordPos) private pure returns (bytes32 slot) {
    assembly {
        slot := add(self.slot, add(wordPos, 1))
    }
}
```

If `wordPos` has dirty high-order bits, the result of the `add()` is incorrect.

Conversely, assembly access of narrow types is done correctly in function `deriveMapping()`, taken from OpenZeppelin, where the high order bits are cleared in the first `mstore`:

```
// copy from @openzeppelin/contracts/utils/SlotDerivation.sol but change to OrderId
function deriveMapping(bytes32 slot, OrderId id) internal pure returns (bytes32 result) {
    assembly ("memory-safe") {
        mstore(0x00, and(id, shr(192, not(0))))
        mstore(0x20, slot)
        result := keccak256(0x00, 0x40)
    }
}
```

In practice, in the case of `wordSlot()`, the argument `wordPos` is the result of function `tickToPos()`, which does not seem to introduce dirty bits, however this behavior is undefined and compiler specific, and it could change by changing compiler settings or in future versions of solidity.

As an example of the effect of dirty bits, consider the following example: `x` and `y` should have the same value, however the `require` fails in Solidity 0.8.28 when compiling without the `--via-ir` flag:

```
function uint_shrink_expand() public {
    uint256 a = 115792089237316195423570985008687907853269984665640564039457584007913129639935;
    uint128 b = uint128(a);
    uint256 x = uint256(b);
    uint256 y;
    assembly{
        y := b
    }
    // Reverts
    require(x == y, "CHECK");
}
```

Code corrected:

The function `wordSlot()` has been updated to increment value `wordPos` outside of the inline assembly block, so the solidity compiler can clear up any dirty high order bits:

```
function wordSlot(
    TickBitmap storage self,
    uint8 wordPos
) private pure returns (StorageSlot.Uint256Slot storage slot) {
    return _slot(self).offset(uint256(wordPos).inc()).getUint256Slot();
}
```

6.5 Force Cancel Risky Users Can Be Blocked

Design | Low | Version 4 | Code Corrected

CS-BOROS-MKT-029

The function `MarketHubRiskManagement.forceCancelAllRiskyUser()` iterates over all markets a user has entered and cancels their open orders in each market. However, if any of the entered markets has expired, the `MarketEntry.cancel()` function reverts, causing the entire operation to revert.

An attacker could exploit this behavior by intentionally remaining in an expired market, effectively blocking the force cancellation of their open orders. While this creates a denial-of-service condition for

the batch operation, the admin can still call `MarketHubRiskManagement.forceCancel()` on each non-expired market individually, so the overall impact is limited.

Code corrected:

In Version 5, the `forceCancelAllRiskyUser()` function has been updated to take a list of markets as a parameter, instead of iterating on all the entered markets of the user. This allows the admin to filter out reverting markets.

6.6 Fee Can Make Liquidations Unprofitable

Design | Low | Version 1 | Code Corrected

CS-BOROS-MKT-004

It is likely that the fee applied to the liquidator makes liquidations as a whole unprofitable, for realistic parameter selection.

The liquidation incentive earned by the liquidator is defined as:

$$|S|\ K_{MM}\ \max(r, I_r)\ \max(\Delta t, minT)\ \min(h, b + s(1-h))$$

and the fee paid by the liquidator is:

$$|S|\ \Delta t\ f_{otc}$$

Liquidations are therefore profitable if

$$K_{MM}\ \max(r, I_r)\ \max(\Delta t, minT)\ \min(h, b + s(1-h)) \geq \Delta t\ f_{otc}$$

For realistic parameters choice we can use:

$K_{MM} = 5\%$

$I_r = 10\%$

$f_{otc} = 0.05\%$

$\min(h, b + s(1-h)) = 10\%$

With this choice of parameters, assuming $\Delta t \geq minT$ we can divide both sides by Δt, and have the inequality evaluate to:

$5\% \cdot 10\% \cdot 10\% = 0.05\%$

which is comparable to the fee rate that needs to be paid by the liquidator. Considering that we are assuming the best conditions for the liquidator to trade, that is the market rate is actually the mark rate and not worse. In practice, this might be a too restrictive assumption, and the market rate could be worse than the mark rate. The fee is an additional factor that might hamper timely liquidations.

Code corrected:

In Version 2, the liquidator is charged a Liquidation `feeRate` instead of the OTC fee. The admin can modify the liquidation fee to ensure that liquidations remain profitable.

6.7 Main Account Has Priority When Increasing Collateral

Design | Low | Version 1 | Code Corrected

The subaccount system allows a single address (`root`) to control multiple market accounts. For a given token id, a user needs to first add the token balance to the main account (`marketId == CROSS, subaccountId == 0`), and then move it to a subaccount (either an isolated `marketId`, or another `subaccountId`). When moving collateral (cash) between accounts, both are checked for the initial margin. This means that if both are below initial margin, but the subaccount (or isolated market account) has a more urgent need of collateral to avoid liquidations, the collateral cannot reach the subaccount because it will be used to satisfy the main account Initial Margin check.

Code corrected:

In Version 2 a user can top up the collateral of any subaccount directly.

6.8 Margin Rounding Allows Exiting With Open Positions

Design | Low | Version 1 | Code Corrected

In Version 1 of the code, the function `_getIMPostProcess()` rounds down the initial margin of a user (code rearranged for legibility):

```
(uint256 long, uint256 short) = user.pending.unpack();
uint256 maxLong = (user.signedSize + int256(long)).abs();
uint256 maxShort = (user.signedSize - int256(short)).abs();
PMath.max(maxLong, maxShort).mulDown(marginFactor).mulDown(marginIndexRate)
```

In particular, the multiplications with `marginFactor` and `marginIndexRate` round towards zero, which could result in an initial margin too low.

For example, with `marginFactor` of 0.1 and a `marginIndexRate` of 1% (0.01), an order of size 999 weis would end up requiring 0 margin.

Since in `MarketHub.exitMarket()`, a margin of 0 is used to verify that a user has no open order in a market, this can be exploited to exit markets while orders are still open, potentially creating few weis of insolvency and breaking invariants.

Code corrected:

In Version 2 the `exitMarket` function now checks for open orders by examining the user's size and the number of open orders.

```
function exitMarket(MarketAcc user, MarketId marketId) external onlyRouter {
    ...
    bool positionIsEmpty = signedSize == 0 && nOrders == 0;
    require(positionIsEmpty || _isMarketMatured(market), Err.MMMarketExitDenied());
```

The change prevents a user from exiting a market while having open orders, even if the margin has been rounded down to zero.

6.9 No Minimum Position Size Requirement

Design Low Version 1 Code Corrected

CS-BOROS-MKT-008

Users can create positions of all sizes including tiny positions of a few wei. The liquidation incentive is proportional to the size of the position, so these tiny positions can be unprofitable to liquidate considering gas costs.

As a result, small positions can get into negative health and then stay unliquidated. This creates bad debt as the losses of these positions are paid out to counterparties and the treasury. An attacker could attempt to exploit this behavior by creating thousands of tiny positions that are unlikely to be liquidated.

However, this strategy will likely not be profitable for the attacker as creating each position comes with a onetime cost, and the potential upside is minimal. As long as maximum leverage remains modest relative to market volatility, the potential percentage gain on any position is limited. In such cases, the absolute profit from a small position is unlikely to justify the gas cost required to claim it.

Code corrected:

In Version 2, the risk of a spam attack is mitigated by introducing two mechanisms to make spam costly for attackers:

1. Traders must possess an initial cash balance in their account when entering a market, increasing the capital requirements for attackers.
2. Traders are charged a market entrance fee when entering a new market, raising the cost of creating each position.

6.10 PREVRANDAO Opcode Behavior on L2 Chains

Correctness Low Version 1 Code Corrected

CS-BOROS-MKT-009

The function `SweepProcessUtils.__sweepFOneSide()` uses the *PREVRANDAO* opcode as a source of randomness for the quicksort and random partition algorithm. On Ethereum Mainnet, this opcode returns the previous *RANDAO* value. However, the protocol is expected to be deployed on Base Chain or Arbitrum, where opcode has different behavior:

- **Arbitrum**: The opcode returns the constant value *1*.
- **Base Chain**: The opcode returns the previous *RANDAO*` value from the latest synced L1 state, which may remain unchanged for multiple blocks.

This makes the randomization of the algorithms predictable, and an attacker could try to create orders that must be swept with the worst-case performance of the quicksort and random partition search, to increase the gas cost of a liquidation or of a partial match.

Code corrected:

The function `SweepProcessUtils.__sweepFOneSide()` has been updated to use the *block.number* as a source of randomness instead.

6.11 `forceDeleverage()` Does Not Follow Specification

Design | Low | Version 1 | Specification Changed

CS-BOROS-MKT-011

The Whitepaper specifies that forced deleverage (paragraph 3.3.11) should have a healthy winner and an unhealthy loser. This is not enforced in the code for `forceDeleverage()`, which applies no restrictions on the health of winner and loser.

Code corrected:

Pendle states that the checks will be implemented in an external deleverager contract.

6.12 `signedSize` Internal Method Returns Incorrect Result

Correctness | Low | Version 1 | Code Corrected

CS-BOROS-MKT-013

The `signedSize()` function defined in `MarketTypes.sol` for type `AccountData2` returns an incorrect value: it simply returns the 128 low order bits of `AccountData2` as a signed integer, but the `signedSize` is actually stored in the bits `191..64` of `AccountData2`.

The function is unused, so the overall system is unaffected.

Code corrected:

The function has been removed in Version 2.

6.13 Finalize Withdrawal Violates CEI Pattern

Informational | Version 2 | Code Corrected

CS-BOROS-MKT-025

The function `MarginManager.finalizeVaultWithdrawal()` sends tokens to the user before setting the user's balance to zero. This order of the operations violates the Checks-Effects-Interactions (CEI) pattern, which requires that state changes are made before any external calls. If the token used would be reentrant (i.e. ERC-777), an attacker could re-enter the margin manager and withdraw their balance a second time. In practice this is not a security concern, as the system does not allow reentrant tokens.

Code corrected:

The code has been updated to follow CEI pattern.

6.14 Inconsistent Specification

Informational | Version 2 | Code Corrected

CS-BOROS-MKT-026

Code comments in the `UserResult` and `OTCResult` structures, in file `MarketTypes.sol`, imply that `UserResult.finalVM` will be zero when `isStrictIM` is false and no strict margin check is performed:

```
VMResult finalVM; // if isStrictIM is true, then finalVM is finalIM, if strict then it's MM, else it's ZERO
```

However, ZERO is never returned.

Code corrected:

The code comments in `UserResult` and `OTCResult` have been corrected.

6.15 Misnamed Variables and Functions

Informational | Version 1 | Code Corrected

CS-BOROS-MKT-016

Variable `maxMarginIndexRate`, which defines the minimum absolute rate that is used in the margin value computation, should instead be named `minMarginIndexRate`, since it is a minimum.

Internal function `canSettle()` returns true even if the order is already settled or cancelled.

Function `tryRemove()` reverts with error `MarketOrderFilled()` when called with flag `isStrictCancel` set to true. The error name implies the order has already been filled, but it might have instead already been cancelled.

In function `_settleProcessGetHealth()` of `MarginManager`, local variable `totalIM` represents actualy a maintenance margin (MM), and not an initial margin (IM). Likewise, comment in function `_settleProcess()` of `MarketHub` mentions `totalIM`, even though it could be a maintenance margin. Function `__settleProcessGetIM()` can likewise also return maintenance margin, and in its body the specific "IM" is used for variable naming even though the variables might represent also maintenance margins.

Code corrected:

All the aforementioned variables and function have been renamed in Version 2.

6.16 `_getRateAtTick` Does Not Function Over Its Whole Domain

Informational | Version 1 | Specification Changed

CS-BOROS-MKT-019

Internal function `_getRateAtTick()` accepts an `int24` argument, but it does not return correct results over the whole `int24` range (for example it returns 0 for `tick == 2**19`).

The function however returns correct results over the range

$tick \in [-2^{int16.min * 15}, 2^{int16.max * 15}]$

which is the rate over which this function is used within the codebase. The valid range should be documented in case the max tick spacing ever gets increased, or if the function gets reused in other contexts.

Specification changed:

Pendle has added code comments to the function describing the valid range of ticks over which the function returns correct results.

6.17 `coverLength` of Maximum Index Node Is Incorrect

Informational | Version 1 | Code Corrected

CS-BOROS-MKT-021

Node ids are `uint40`, meaning the greatest possible value for a node id in the fenwick tree of orders is `2**40 - 1`. This node should have cover length of 4**9, however function `coverLength()` returns 0 when `2**40 - 1` is passed as input.

Code corrected:

In Version 2, the function `coverLength()` has been updated to return the correct value for the maximum node id.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Force Deleverage and Liquidate Can Be Frontrun

Informational | Version 1 | Acknowledged

CS-BOROS-MKT-005

The function `forceDeleverage()` can be frontrun by reducing the size of the winning or losing address, causing the `reduceOnly` check to fail and the transaction to revert. The `reduceOnly` check in the function `_isReducedOnly` ensures that the size of the forced trade is smaller than the current size and that the sign is opposite. This check guarantees that the position is strictly reduced and prevents the trade from flipping the size's sign (e.g., turning a long position into a short position or vice versa).

```
function _isReducedOnly(int256 curSize, int256 newTradeSize) internal pure returns (bool) {
    return newTradeSize.abs() <= curSize.abs() && newTradeSize.sign() * curSize.sign() <= 0;
}
```

An attacker can monitor the mempool for deleveraging transactions. Upon detection, the winning or losing address could reduce their position size below the size of the trade, causing the trade to flip the position's sign and the transaction to revert.

Similarly, the function `liquidate()` can also be frontrun by an attacker. By decreasing their position size, the attacker can ensure that the size of the liquidation trade exceeds the signed size of their position, causing the transaction to revert due to the same `_isReducedOnly` check. While the strong margin check typically prevents a liquidatable user from modifying their position, the weak margin check allows an action as long as it improves their position.

This behavior poses a risk to the platform's risk management as liquidations and force deleveraging are delayed. However, since the protocol is expected to be deployed on an L2 without a public mempool, the likelihood of frontrunning is considered low. An attacker could still decide to change their position every block to block liquidations, but this would be very costly.

This risk is particularly problematic for callers unaware of this behavior. A more sophisticated caller could programmatically retrieve the current size of the user's position within the transaction and cap the size of the trade with it.

Acknowledged:

Pendle answered:

> Key reason being Morpho & Compound also don't take max of this number with users' outstanding debt. The expectation of an external contract calling & pass in the correct amount is rather reasonable.

7.2 Gas Optimizations

Informational | Version 1 | Code Partially Corrected

CS-BOROS-MKT-014

Version 1:

1. Function `MarketOrderAndOtc._hasSelfFilledAfterMatch()` could only check the side the user has previously matched with.
2. Functions `extend()`, `concat()`, `sliceFrom()` of `ArrayLib.sol` and function `makeTempArray()` of `LibOrderIdSort.sol` allocate empty arrays with the `new` keyword. This generates code that will zero the newly allocated array, which is wasteful since the array immediately gets written to, in the aforementioned functions.
3. In `MarketEntry.liquidate()`, `_coreRemoveAll()` is called before sweeping the violator's orders. Removing after sweeping could be more efficient, as every removed order is a storage access, while the storage complexity of sweeping is sublinear in the number of orders.
4. In `MarketOrderAndOtc._matchOrder()`, at line 106 we sweep all OTC counterparties after matching on the order book. Both sides will be swept for every counterparty, while it would be enough to sweep the side where we are matching.
5. In `__matchPartialInner()`, in line 192 `subtreeSum` is set to 0 even for leave nodes.

Version 4:

6. Function `CoreOrderUtils._shouldPlaceOnBook()` could return `!hasMatchedAll` for `SOFT_ALO`, allowing `_coreAdd()` to return early if all orders are removed.

Code partially corrected:

The code optimizations 1, 2, 3, 4 and 5 have been implemented.

7.3 Margin Rounds Down

Informational | Version 1 | Acknowledged

CS-BOROS-MKT-028

The function `_getIMPostProcess()` rounds down the initial margin of a user and can allow a user to have possibly too much leverage on positions with very low sizes, potentially creating creating dust amounts of bad debt.

Acknowledged:

Pendle has provided the following reasoning for their design decision:

> In the current codebase, the rounding directions for all value types are correct (e.g. fees rounded up, payment rounded down), except for PM which is rounded down. We chose not to round PM up as it would complicate the code quite a lot (partial PM must be computed exactly, PM can no longer be computed from trade cost). However, that means size = 1 would have MM = 0 -> health ratio can not be calculated for liquidation.

7.4 Packed Variables in Events

Informational | Version 1 | Acknowledged

CS-BOROS-MKT-017

Events `FIndexUpdated` has a argument of type `FIndex`, which is a human unreadable packed type. Likewise, `ForceDeleverage`, `MarketOrdersFilled`, `OtcSwap`, `Liquidate` include arguments of

type `Trade` which is unreadable when emitted. This can add friction when implementing front-ends and debugging.

Acknowledged:

The team is aware of this behavior, but has decided to keep the code unchanged.

7.5 Unnecessary Unchecked Block in `makeTempArray()`

Informational **Version 1** **Acknowledged**

CS-BOROS-MKT-018

Internal function `makeTempArray()` wraps its body in an `unchecked` block. However, the only arithmetic operation is the `++i` increment in the iterator of the loop:

```
for (uint256 i = 0; i < n; ++i) {
    arr._set(i, OrderIdEntryLib.from(ids[i], i));
}
```

Loop increments of that form are by default unchecked since solidity `0.8.22` (link to docs). So, the unchecked block is not necessary.

Acknowledged:

Pendle is aware of this behavior. They nonetheless choose to keep the code unchanged for consistency with the rest of the codebase.

7.6 Unpacking OrderId Can Access Dirty Bits

Informational **Version 1** **Acknowledged**

CS-BOROS-MKT-027

The structure `OrderID` is stored as a 64-bit value and contains several fields, including an initialized marker, reserved bits, side, tick index, and order index.

The function `OrderIdLib.unpack()` extracts the side, tick index, and order index from the structure but ignores the initialized marker and reserved bits. This can lead to multiple `OrderId` values being unpacked into the same side, tick index, and order index values.

In most cases, `OrderID` is unpacked only after being created by `OrderIdLib.from()`, which ensures the reserved bits are empty, and the initialized marker is set. However, the function `getOrder()` takes any `OrderId` as argument and then unpacks it without validating the initialized marker or reserved bits. So an invalid or non-existent `OrderId` values could to be interpreted as another valid `OrderId`. Although this function is not actively used within the protocol, integrators could call this function with an invalid `OrderId` and cause unexpected behavior.

Acknowledged:

Pendle answered:

We acknowledge that some bits in OrderId are not used at all. We decided not to validate those bits when unpacking user-provided OrderIds because it increases gas with no security benefits.

7.7 `cashTransferAll` Executes the `onlyAllowed` Modifier Twice

Informational **Version 1** **Acknowledged**

CS-BOROS-MKT-020

Function `cashTransferAll()` of `MarginManager` has the `onlyAllowed` modifier, and internally calls the `cashTransfer()` function, which also has the `onlyAllowed` modifier. The modifier is therefore executed twice.

Acknowledged:

The Pendle team is aware of this behavior, but has decided to keep the code unchanged.

7.8 `getOrderStatusAndSize()` Does Not Distinguish Cancelled From Non-Existing Orders

Informational **Version 1** **Acknowledged**

CS-BOROS-MKT-022

View function `getOrderStatusAndSize()` could return `CANCELED` for orders that have been cancelled. However, it just returns `NOT_EXIST` for both non-existing orders and cancelled orders.

Acknowledged:

The Pendle team is aware of this behavior, but has decided to keep the code unchanged.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Bulk Orders Enable Temporal Undercollateralization

Note Version 2

Bulk orders allow accounts to place orders across multiple markets and in both the long and short direction within a single call. Unlike individual orders batched together in a multicall, the margin requirements are not checked after each order is processed, but only after all bulk orders are processed.

For example, with two bulk orders B1 and B2, the following steps are executed:

1. Match and add B1.
2. Process payments and fees for B1.
3. Match and add B2.
4. Process payments and fees for B2.
5. Check margin requirements for account.

Since the margin requirements are only enforced after bulk order B2 is processed, the account could be temporarily undercollateralized after processing B1. This can be compared to a flashloan and makes certain operations more capital-efficient. For instance, accounts can simultaneously match against both sides of the orderbook with reduced collateral requirements, since opposing positions can be netted against each other.

8.2 Gas Considerations on Algorithm Complexity

Note Version 1

It is important for the correct function of markets that the settle operation uses a bounded and reasonable amount of gas. Settle is required before matching, liquidations, force closes and deleverages, and an attacker who is able to make settlement run out of gas can cause denials of service and insolvency risks to markets. It is therefore necessary tha the gas requirements of settling are predictable and bounded.

Settling a user in a given trading zone (specific `tokenId`) requires settling the user in each of the markets they have entered. The number of iterations for this loop is bounded by `MAX_ENTERED_MARKETS`. `MAX_ENTERED_MARKETS` should therefore not be set too high. Settling a user in a specific market requires iterating over all of their open orders, and individually checking if they have been filled. The number of iterations for this loop is bounded by `MarketCtx.maxOpenOrders`. `maxOpenOrders` should therefore not be set too high. The fTime of filled orders needs to be found. This performs a binary search that takes at most `log2((maturity - marketCreationTime) / period)`. In practice this is below 23 iterations for a market with 10 years maturity and fIndex update every second.

Substantially, the cost of settling a user scales linearly with `MAX_ENTERED_MARKETS * maxOpenOrders`. This value should be therefore set appropriately, to avoid the possibilty of Denials of Service attacks.

8.3 Pre-scaling Margin Collects Dust

Note Version 2

When a user places an order in the market, the pre-scaling margin for the new order is calculated in `__calcPMFromRate()` based on the size and rate of the order. Depending on the direction of the order, the calculated margin is added to either *sumLongPM* (for long positions) or *sumShortPM* (for short positions).

```
function __calcPMFromRate(uint256 absSize, int256 rate, uint256 k_iThresh) private pure returns (uint256) {
    uint256 absRate = rate.abs();
    return absSize.mulDown(absRate.max(k_iThresh));
}
```

When the order is (completely) filled or canceled, the reverse operation occurs: the pre-scaling margin is recalculated and deducted from `sumLongPM` or `sumShortPM`. Since the operations are symmetrical, the pre-scaling margin after the order is removed matches the pre-scaling margin before the order was added.

However, in the case of partial fills, the pre-scaling margin is calculated based on the partially filled size of the order. Note that the margin is rounded down during calculations, so it may be up to 1 wei smaller. This means that each time an order is partially filled, a rounding error of 1 wei may occur, causing a slight discrepancy compared to when the order was first added. Over time, these small rounding errors can accumulate, leaving a small residual "dust" in the pre-scaling margin once the order is completely filled.

While this rounding error is small and unlikely to significantly affect margin calculations, it is important for integrators of the protocol to be aware that the pre-scaling margin may contain small amounts of dust. Users should not expect the margin to exactly match the size of the orders placed.

Note than when all orders for a user are filled, the values for `nLongOrders` and `nShortOrders` are reset to zero, and the pre-scaling margin is no longer used in the `_initUserCoreData` function. This effectively removes any residual dust from the pre-scaling margin. So users can to reset their pre-scaling margin to zero by canceling all their standing orders.

8.4 Security Considerations of Sequencer Outages

Note Version 1

Pendle V3 is expected to be deployed to an Ethereum L2, such as Base or Arbitrum. These networks rely on a centralized sequencer to order transactions and produce blocks. If the sequencer goes offline, the L2 becomes unable to process transactions in a timely manner.

In such scenarios, users can bypass the sequencer by using the force inclusion mechanism, submitting L2 transactions directly on Ethereum Mainnet. However, this method is costly and requires technical expertise, making it inaccessible to most users. If trading is unrestricted then users may be unable to cancel their trades, allowing others to fill their orders at stale prices. Additionally, reduced trading activity during a sequencer outage can make the mark price, which is based on a TWAP (Time-Weighted Average Price), less reliable. This could lead to unfair liquidations, even for users whose positions would otherwise remain healthy.

There exist oracle solutions to detect sequencer outages, but none are implemented in Version 1. Future versions of the Boros Markets could integrate an oracle to monitor for outages and restrict certain user actions during these periods. However, if trading were restricted then the TWAP does not get updated anymore and positions could become unhealthy and create bad debt as they cannot get liquidated fast enough.