REDACTED

Code Assessment

of the Boros Router and AMM Smart Contracts

07 August, 2025

Produced for

PENDLE

by

CHAINSECURITY

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 Router and AMM according to Scope to support you in forming an opinion on their security risks.

Pendle implements Boros Router and AMM. The Router is the entry point for users interacting with the Boros system, an onchain marketplace for Interest Rate Swaps. The AMM allows liquidity providers to passively provide liquidity to Interest Rate Swap markets.

The most critical subjects covered in our audit are functional correctness, solvency, and access control. Security regarding all the aforementioned subjects is high.

Lastly, some low and medium severity issue have been redacted to prevent malicious actors from creating disturbances. Pendle has either mitigated or accepted all the risks.

In summary, we find that the codebase provides a good level of security.

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	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	5
• Code Corrected	3
• Specification Changed	1
• Risk Accepted	1

Please note that upon request of the customer certain findings have been redacted from this report.

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 Router and AMM 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	14 April 2025	5a12496fa73637e85b73cbcd1a141737782854fb	Initial Version
2	25 May 2025	4f769483ded7dff9afd226bba9e0e171c8e95a26	Second version
3	12 June 2025	8f9855c22ae4a1c733de8756e7ab26f6c7c7b7d3	Third version
4	20 July 2025	e2aa73d0c4b8d427bed661ee0a747b1d8e11aa6c	Forth version
5	25 July 2025	a905b3788f13edba3bbdb7e49b5594ae51f28b31	Fifth 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/amm/NegativeAMM.sol
./contracts/core/amm/PositiveAMM.sol
./contracts/core/amm/BaseAMM.sol
./contracts/core/amm/PositiveAMMMath.sol
./contracts/core/amm/BOROS20.sol
./contracts/core/amm/NegativeAMMMath.sol
./contracts/core/router/trade-base/BookAmmSwapBase.sol
./contracts/core/router/trade-base/TradeStorage.sol
./contracts/core/router/math/LiquidityMath.sol
./contracts/core/router/math/TickSweepStateLib.sol
./contracts/core/router/math/SwapMath.sol
./contracts/core/router/Router.sol
./contracts/core/router/auth-base/AuthBase.sol
./contracts/core/router/auth-base/EIP712.sol
./contracts/core/router/auth-base/SigningBase.sol
./contracts/core/router/auth-base/RouterAccountBase.sol
./contracts/core/router/modules/AuthModule.sol
./contracts/core/router/modules/TradeModule.sol
./contracts/core/router/modules/MiscModule.sol
./contracts/core/router/modules/AMMModule.sol
./contracts/core/roles/PendleRoles.sol
./contracts/core/roles/PendleAccessController.sol
./contracts/lib/Errors.sol
./contracts/lib/PaymentLib.sol
./contracts/types/Account.sol
./contracts/types/MarketTypes.sol
./contracts/types/Trade.sol
```

In Version 4 the following contract was added to the scope:

```
./contracts/core/router/auth-base/AuthStorage.sol
```

2.1.1 Excluded from scope

Any contracts that are not explicitly listed above are out of the scope of this review. Namely, third-party libraries are explicitly out of the scope of 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, an Interest Rate Swap marketplace featuring leverage, an orderbook exchange, and AMMs. This report focuses on the periphery of the system: the Router and the AMM.

2.2.1 The Router

The role of the Router is to perform access control, interact with the core system and AMMs on behalf of users, and provide some utility methods to route trades through the Orderbook and the AMM in a balanced and price-efficient way.

The implementation of the Router loosely follow the diamond pattern, with a central `Router` contract that delegatecalls modules for specific functionality:

- *AuthModule* implements signature based authentication, and access delegation to agents.
- *TradeModule* handles interactions with the trading functionality, and handles balance and token transfers.
- *AMMModule* handles liqiuidity provision and swapping with the AMMs.
- *MiscModule* implements varias such as EIP712 initialization, simulation functionality, and administrative functions to set router parameters.

Contrary to the usual implementation of the diamond pattern, where implementation contracts are selected based on a storage mapping from selector to implementation, Boros Router selects the implementation contract for a given selector based on a hardcoded search tree. Addresses and function selectors are hardcoded and not configurable once the Router is deployed. However, the Router is expected to be deployed behind a Transparent Upgradeable Proxy.

2.2.1.1 AuthModule

The *AuthModule* implements account management and access delegation for users of the MarketHub and AMM. All functions in the *AuthModule* are protected by the `onlyRelayer` modifier, allowing only whitelisted relayer contracts to call them. At the core of the Boros account model is the user's *Account*, which is the user's EVM address plus one byte of *subaccountId*. A user can set an account manager for an account they control, by signing (either with ECDSA for EOAs or EIP1271 for smart wallets) an appropriate message which is used in function `AuthModule.setAccManager()`.

The *AccountManager* and the user themselves has the power to approve or revoke *agents*. An *Agent* can sign messages to execute trading and AMM functionality on behalf of the user. The idea is that a user needs not to interact with their crypto wallet for every action that is performed on Boros, but enables a key to sign on its behalf inside the Boros web application. Actual transactions relaying these signatures on-chain are expected to be sponsored by Pendle. The *Agent* has less privileges than the actual user,

and it can use a subset of *non-destructive* functionality: in particular, an agent cannot withdraw a user's balance from Boros or pull funds from the user. Actions signed by Agents can be executed through function `agentExecute()`, which sets the currently authenticated user in transient storage, and then executes the desired function by delegatecalling the *Router*. When setting an Agent, an expiry time can be set to limit its privileges.

Users can access router functionality directly without agents. When router functions with the `setNonAuth` modifier are accessed, the Router checks if an authenticated user is set in a specific transient storage slot. If it is, it means execution was launched by an agent, and the authenticated account is retrieved from transient storage. In the other case, the default Account derived from `msg.sender` (*SubaccountId* zero) is used.

The *AuthModule* also contains functionality to deposit into Boros and request fund withdrawals through messages signed by the users directly (not their agents) with methods `vaultDeposit()`, `requestVaultWithdrawal()`, `cancelVaultWithdrawal()`, `subaccountTransfer()`, and `vaultPayTreasury()`.

2.2.1.2 TradeModule

The *TradeModule* implements interactions between users and the MarketHub, the Orderbook exchanges, and the AMMs.

It implements deposit functionality in function `vaultDeposit()`: A user can deposit into their account a given amount of a token, specified by its *tokenId* in the MarketHub.

Functions `subaccountTransfer()` and `cashTransfer()` allow to respectively move tokens between the main cross account and a subaccount (denoted by a *subaccountId* and its *marketId*), and from a cross account (meaning that it interacts with cross-margined markets) and an isolated account (interacting with only one market).

Functions `requestVaultWithdrawal()`, `cancelVaultWithdrawal()`, and `finalizeVaultWithdrawal()`, allow the main cross account for a given *tokenId* to withdraw collateral. After being *requested*, the withdrawal is delayed, and after the delay it can be *finalized* by anyone.

Function `payTreasury()` allows donating to the treasury, in order to contribute to sponsored transactions.

Function `placeSingleOrder()` allows placing, and potentially immediately filling, an order to buy or sell a given size on a specified market. The order specifies a limit rate, a time-in-force, and optionally an `ammId`. If the order includes an `ammId`, the order is not only matched against the order book, but optimally routed between the orderbook and the specified AMM. A set of ancillary actions can moreover be performed by `placeSingleOrder()`:

- enter the market before placing the order.
- exit the market after filling the order.
- cancel an existing order, before placing the new one.
- move an amount of cash into the account, before placing the order, if an isolated account is being used.
- move all cash out of the isolated account, if an isolated account is used, after placing the order.

Slippage protection is implemented by checking if the order amount that's immediately filled satisfies setting a minimum (or maximum) rate when selling (or respectively buying).

When using `placeSingleOrder()`, the order can be immediately filled, partially filled, or simply be added to the orderbook. Function `bulkOrders()` allows placing (and potentially matching) multiple orders at once. It also allows cancelling a list of orders, and if any of the orders to cancel is not found, or already cancelled, it reverts.

Function `bulkCancels()` allows soft-cancelling a list of orders (if they are already filled, the function does not revert). `enterExitMarkets()` allows entering/exiting multiple markets at once. `liquidate()` allows liquidating unhealthy positions of other users, in practice it simply performs access control before forwarding the call to the *MarketHub*. Finally, there's function `settlePaymentAndOrders()` that settles all the markets that an account has entered and synchronizes them with the MarketHub. This last function is unauthenticated.

2.2.1.3 AMMModule

The *AMMModule* implements interactions between users and the AMMs. *AMMModule* provides functions to supply liquidity to an AMM, through `addLiquidityDualToAmm()` and `addLiquiditySingleCashToAmm()`. Function `addLiquidityDualToAMM()` supplies liquidity in a balanced way, adding cash and size in the current proportions they are held in the AMM. It allows a user to specify the *size* to supply to the AMM, and derives the corresponding cash amount that is also pulled from the liquidity provider. Two slippage protection parameters are available: `maxCashIn` and `minLpOut`.

Function `addLiquiditySingleCashToAmm()` allows the liquidity provider to supply a cash amount, and will trade it to variable rate size such that the *size* acquired and the *cash* left can be added to the AMM in the current AMM proportions. Finding the amount of size that needs to be acquired for balanced supplying requires an iterative search process, to find the cost to acquire a given size through the orderbook and AMM, and what is the correct ratio to add cash and size after the swap, which changes the AMM ratio. This function internally uses similar logic to the swap routing between AMM and orderbook present in `placeSingleOrder()` to perform the search.

Both functions to add liquidity have a `minLpOut` slippage protection parameter. Its use is essential to avoid possible sandwich attacks on liquidity providers. Indeed, AMMs allow inflating the value of LPs when trading above or below the actual market price. The LP token price is then brought back to normal when the AMM price is brought to the market price by arbitrage. If LP tokens are minted while the price is inflated, the minter will receive fewer tokens, which will immediately lose value when the price is arbitraged back to normal.

To remove liquidity, there are functions `removeLiquidityDualFromAmm()`, which burns LP tokens and returns size and cash to the user proportionally to the amount in the AMM, and `removeLiquiditySingleCashFromAmm()`, which removes cash and size from the AMM by burning LP tokens, and then trade the size amount for cash on the orderbook and AMM. When removing liquidity *dual* (cash and size), minimum amount for cash can be specified, and a minimum and maximum size amount can also be specified, since the rate in the AMM could be negative (for negative AMM), in that case the user wants to receive a maximum amount of size, but in case rate is positive the user wants to receive a minimum amount of size. The maximum size requirement can be useful to avoid acquiring an unexpectedly big long position (or the minimum size requirement to avoid too big short positions for negative AMM). However, when `removingLiquidityDualFromAmm()` is used slippage protection is less important, since the LP token value can't be deflated by swapping (no sandwiching). For `removeLiquiditySingleCashFromAmm()` slippage protection is applied in terms of `minCashOut`. Since swapping is involved, where slippage can be significant, it is important to specify slippage protection when using `removeLiquiditySingleCashFromAmm()`.

Function `swapWithAmm()` allows swapping cash for variable rate size, or vice versa, through the AMM. It allows specifying the input/output size to be swapped, and a `desiredSwapRate` as a slippage protection parameter.

2.2.1.4 MiscModule

The *MiscModule* contains functions for Router initialization, simple off-chain simulation through RPC calls, and setters and getters for parameters of the Router.

- `initialize()`: is called atomically during *Router* deployment, and allows setting the `name` and `version` of the contract for the domain separator in signatures, conforming to EIP-712. It also allows setting the `numTicksToTryAtOnce` configuration, which is used when iterating over the

order book to route orders and liquidity provision between AMM and Order book. The effect of `numTicksToTryAtOnce` is to tune the gas consumption of functions `placeSingleOrder()`, `addLiquiditySingleCashToAmm`, and `removeLiquiditySingleCashFromAmm()`, which can be useful in case of different liquidity concentrations: trying more ticks at once when the liquidity is more sparse.

- `tryAggregate()` acts as a multicall for the *Router*, calls to router functions can be passed atomically through `tryAggregate()`.
- `batchSimulate()` and `batchRevert()` allow impersonating accounts through the authentication mechanism before calling arbitrary contracts. Any effect of the calls are reverted, so these functions can be called to simulate interactions with the Boros system.
- Setters and getters for the `ammId` to account mapping (the account of the AMM), for configuration parameters of the *Router* such as `numTicksToTryAtOnce`, `maxIterationsAddLiquidity`, and `epsAddLiquidity`.

2.2.2 The AMM

Boros offers an AMM to complement the orderbook exchange for a given Interest Rate Swap market. On the high level, the AMM allows liquidity providers to supply cash and variable rate positions, in exchange for LP tokens. The AMM then allows user to trade cash for variable rate positions and vice versa, while collecting fees for liquidity providers. Differently to orderbook makers, which are active liquidity providers, AMM liquidity providers are mostly passive, and are not expected to frequently interact with the AMM once they have provided liquidity.

The AMM is based on a bounding curve inspired by the Balancer formula $x_x^w y_y^w = k$. The two assets in the AMM are variable rate income stream (x) and fixed rate income stream (y). The variable rate income stream x is denominated in `size`, the same unit of user positions in *Markets*. The fixed rate income stream (y) is denominated in units corresponding to a 100% APR from the current time, until maturity. For a given time to maturity T denominated in years, the fixed rate income stream for some cash amount c is such that $y = c/T$. In the context of the AMM, a variable t representing the advancement of the market towards maturity is used: it starts at 1 when the AMM is created, and linearly goes towards 0 when maturity is reached. It is based on the latest `fTime` queried from the *Market*.

$$t = \frac{T}{D}$$

where T is time to maturity, and math:D is the time from AMM creation to maturity. The quantity y, of fixed income stream, changes as the market progress, since a cash amount c covers for more fixed income stream. However, the quantity yt does not change as t changes, since:

$$yt = y\frac{T}{D} = \frac{c}{D}$$

which is constant.

A "virtual" amount of variable income stream is added to the actual AMM balance, so that the actual AMM balance can become negative, meaning the AMM can take short positions when selling variable income stream, which improves the capital efficiency of the AMM. To do so, the AMM tracks the amount

The AMM invariant used for swapping is:

$$(x + a)^t y = k$$

The price (fixed rate) defined by this invariant is (taking the derivative of y w.r.t. x):

$$p = \frac{yt}{x + a}$$

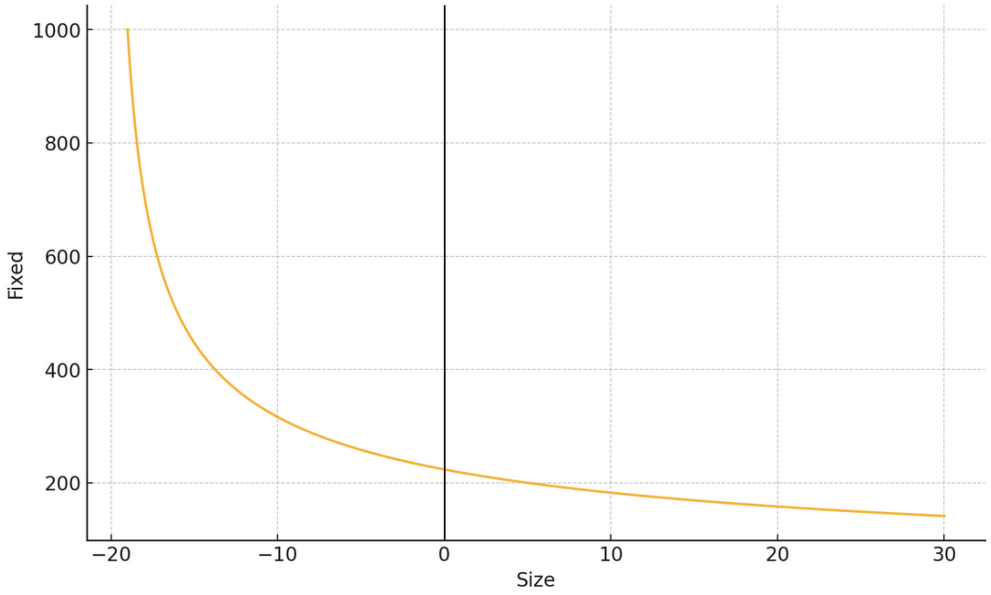
We know that yt is constant as t changes, so the price in the AMM does not change, even if the y amount does, as time progresses. Beside x and y, the AMM also controls an amount B of cash, called buffer, that it uses as additional collateralization for its positions. Fees are collected in cash on every swap, and they accrue in the buffer:

$f_{amm} = |\Delta x| r_{fee} T$

When supplying liquidity, the AMM stops operating in case it undergoes liquidations or force deleveraging, but still allows Liquidity Providers to withdraw their liquidity.

Bounds are defined on the trading range by `minAbsRate` and `maxAbsRate`. The price after a swap must be between those bounds. It is also possible that implicit bounds are defined by failing initial margin checks after a swap.

The formula showed above for the AMM only supports positive prices (rates). For every market a negative AMM can be defined by flipping the AMM design around the y axis. x amounts are then inverted, and so are rates. This allows trading negative rates on a separate AMM. When interest rates are very close to 0, the AMM becomes unpractical to trade interest rate swaps.



2.2.3 Differences between Version 1 and Version 2

In Version 2 the deposit and withdrawal logic is adapted to enable interacting with the newly introduced withdrawal delay.

A total supply cap is introduced in the AMM, to prevent the AMM position from becoming too big and potentially dangerous.

2.2.4 Differences between Version 3 and Version 4

In Version 4, all functions in the *AuthModule* are now only callable by whitelisted relayers, whereas in previous versions the module was intended to be called by anyone.

Users can now approve or revoke agents directly instead of going through an account manager.

Upgradeable contracts are now deployed behind a Transparent Upgradeable Proxy (TUP) instead of a Universal Upgradeable Proxy (UUPS).

2.3 Trust Model

The MasterAdmin is fully trusted. They can upgrade contracts. As the Router is upgradeable and is the only entrypoint for users, users fully trust the MasterAdmin.

Each AMM has an *ammAdmin* which can set parameters that affect swapping, but should not be able to prevent liquidity providers from withdrawing their liquidity.

All contracts are upgradeable except for AMMs, which are not upgradeable, but the AMMFactory factory is.

Users utilizing the `AuthModule` must fully trust the whitelisted relayers to remain operational at all times. Relayer unavailability can result in denial of service for users depending on them. More critically, users with subaccounts face the risk of permanently losing access to their funds if relayers shut down, as subaccounts can only be accessed through relayers, as discussed in Trust model of subaccounts.

Agents and AccManager are trusted by users.

Tokens used do not implement callbacks on receive, are non-rebasing and do not have fees on transfer.

The system is expected to be deployed on Base or Arbitrum. The L2 sequencer is trusted to process transactions in the order received.

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	1

- Backrunning Stale Prices on Findex Updates Risk Accepted

5.1 Backrunning Stale Prices on Findex Updates

Design Low Version 1 Risk Accepted

CS-BOROS-AMM-003

The price of the AMM (referred to as the "implied rate") is defined as:

$$p = \frac{yt}{x + a}$$

This price remains unchanged when the AMM is not trading. Specifically, it does not change when the t is updated during a *FIndex* update, as yt remains constant.

During each *FIndex* update, floating rate payments are exchanged between short and long traders. Hence, the price of the floating token can be expressed as the discounted value of future floating payments minus the fixed leg. With each update, floating payments are realized, which leads to the following effect on the price:

1. If the floating payment is lower than the average payment, the price should increase after an update.
2. If the floating payment is higher than the average payment, the price should decrease after an update.

To illustrate this behavior, consider a simple example with two periods and an expected floating rate of 0% in period 1 and 10% in period 2 and no fixed payments. Ignoring any discounting and fixed payments, the AMM price can be calculated as the average of the two periods:

$$price = \frac{0\% + 10\%}{2} = 5\%$$

After the first payment of 5% is due, the AMM price would jump to 10%:

$$price = 10\%$$

This behavior creates a scenario where the AMM price becomes stale relative to the market price after the first *FIndex* update. An arbitrageur can exploit this by backrunning the *FIndex* update, trading the AMM back to the market price.

Under normal conditions, the impact of each *FIndex* update is small compared to the total value of the position. However, as the market approaches maturity, the effect of each update can become significant.

Client accepts the risk:

Pendle states:

> We acknowdge that the AMM quoting the same price after an FIndex update is not the most optimal quoting strategy as a market maker. The AMM fees will already mitigate this issue (basically its a spread to protect the AMM from quoting suboptimal prices), which we will be prepared to increase if needed (say when we see a very low/high funding rate settlement incoming, or when we observe toxic arbitrage behaviors). The most harmful way to arbitrage is to sandwich the FIndex update and buy/sell immediately from the AMM, which will be prevented by us controlling the FIndex update transaction. Otherwise, there's no guaranteed profitable arbitrage (because other people can frontrun/backrun your trades). Completely resolving this will involve changing the AMM formula to take into account the small repricing after each FIndex update, which we will only do in future iterations of the AMM and not this version.

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	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	4

- Missing Event in PendleAccessController Code Corrected
- Initial Margin Check Can Block AMM Withdrawals Risk Accepted Specification Changed
- Rate Oracle of AMM Might Return Invalid Values Code Corrected
- ammId in placeSingleOrder Can Mismatch marketId Code Corrected

Informational Findings	4

- Incorrect Specification of AuthModule Code Corrected
- Inconsistent Initialization of Parent Contracts in PendleAccessController Code Corrected
- Typo Code Corrected
- tweakUp Output Rounds Down for Small minAbsRate Code Corrected

6.1 Missing Event in PendleAccessController

Design Low Version 2 Code Corrected

CS-BOROS-AMM-017

Method `setAllowedRole()` of `PendleAccessController` does not emit an event. Because of the missing event, it will be challenging to reconstruct the configuration of the permission system.

Code corrected

Event `AllowedRoleSet` has been added to the method `setAllowedRole()` of `PendleAccessController` in Version 4.

6.2 Initial Margin Check Can Block AMM Withdrawals

Design Low Version 1 Risk Accepted Specification Changed

CS-BOROS-AMM-006

If the mark rate changes such that an AMM fails the initial margin check, liquidity providers are prevented from withdrawing their liquidity.

This can happen for example if an interest rate goes negative: then the positive AMM has a big positive size (which has now negative value) and some amount of cash between the bonding curve and the buffer. Liquidity Providers then might want to withdraw their liquidity before the value decreases additionally, possibly risking liquidation. However, if the mark rate is low enough that the IM check start failing for the AMM, LPs are unable to withdraw, since the OTC swap that follows withdrawing triggers an initial margin check on the AMM.

Client accepts the behavior and states:

> In practice, we will set parameters such that when the AMM fails Initial Margin check, the market is already trading beyond minAPY or maxAPY, and the AMM is already being considered for deleveraging (since the AMM's position is likely big enough to be deleveraged at this point). Users not being able to withdraw at this point is expected.

6.3 Rate Oracle of AMM Might Return Invalid Values

Design | Low | Version 1 | Code Corrected

CS-BOROS-AMM-008

The AMM is bound to only be able to track the rate when its absolute value is between `minAbsRate` and `maxAbsRate`, and before `cutOffTimestamp`. If the rate is outside these bounds, or the timestamp is after `cutOffTimestamp`, functions `impliedRate()` and `oracleImpliedRate()` will silently fail by returning stale values.

Code corrected:

Functions `oracleImpliedRate()` and `impliedRate()` are modified to revert if the market's latest FTime is at or after the `cutOffTimestamp`. The oracles will not update when the market price is below `minAbsRate` and above `maxAbsRate`. This is accepted as intentional behavior.

6.4 `ammId` in `placeSingleOrder` Can Mismatch `marketId`

Correctness | Low | Version 1 | Code Corrected

CS-BOROS-AMM-009

In `TradeModule`, function `placeSingleOrder()` takes as arguments `ammId`, which refers to the AMM potentially used in the swap, and `marketId`, which identifies the market of the swap. However `ammId` can refer to a different market than `marketId`, in which case incorrect order routing can happen.

When `ammId` is non-zero, the marketId used to trade with the orderbook is retrieved from the AMM itself, however, the `marketId` parameter is still used when initializing the `MarketCache` object. The `MarketCache` is passed to internal function `_splitAndSwapBookAMM()`, where it is used to retrieve the `tickStep` of the market, which is accessed in internal function `convertBookTickToBaseRate()`. Since the `tickStep` could be the one of the `marketId` parameter, but the trade is happening on amm.marketId(), which could be a different market, the `tickStep` could be incorrect, and result in incorrect calculation of `withBook` and `withAMM` amounts.

The incorrect amounts returned however do not allow the calling user to extract value from the markets.

Code corrected:

An assertion has been added to ensure that the market of the AMM and the `marketId` of the order are the same.

6.5 Incorrect Specification of AuthModule

Informational **Version 4** **Code Corrected**

CS-BOROS-AMM-019

Code comments in contract `AuthModule` suggest that the module is intended to be callable by anyone.

```
// * Functions in here are callable by anyone, likely Pendle's relayer. No info about msg.sender is used
```

However, all function of the module can be called by whitelisted relayer contracts only.

Code corrected:

In Version 5, the comment has been updated to reflect the code, which only allows Pendle's relayers to call the functions of the module.

6.6 Inconsistent Initialization of Parent Contracts in PendleAccessController

Informational **Version 1** **Code Corrected**

CS-BOROS-AMM-010

In contract `PendleAccessController`, the ancestor contract `AccessControlUpgradeable` is initialized in the `initialize()` method thorugh `__AccessControl_init()`. However, `PendleAccessController` does not inherit directly from `AccessControlUpgradeable`, but from `AccessControlEnumerableUpgradeable`, so `__AccessControlEnumerable_init()` should be called instead. `PendleAccessController` also inherits from `UUPSUpgradeable`, so its initialization function should also be called.

All these actions are however effectively no-ops, included `__AccessControl_init()`, so this is just a matter of consistency and not correctness.

Code corrected:

The initializers of both `AccessControlEnumerableUpgradeable` and `UUPSUpgradeable` are now consistently called.

6.7 Typo

Informational **Version 1** **Code Corrected**

CS-BOROS-AMM-013

Line 3 of `EIP712.sol` contains the incorrectly spelled word `deterministic`.

Corrected:

The typo has been corrected.

6.8 tweakUp Output Rounds Down for Small minAbsRate

Informational | Version 1 | Code Corrected

CS-BOROS-AMM-015

Function `calcSwapSize()` adjusts the minimum absolute rate by tweaking it up by `1e8` before using it in calculation, to prevent hitting the limit and reverting when swapping. However, if the `minAbsRate` is below `1e10`, `minAbsRate.tweakUp(1e8)` will just return the `minAbsRate`, without any uptweak, because internally `tweakUp()` uses `mulDown()` which rounds down. `tweakUp(1e8)` is equivalent to increasing by 1/1e10 of `minAbsRate`, but if `minAbsRate` is below `1e10`, then there is no increase.

Code corrected:

`tweakUp()` has been modified to use `mulUp()` which rounds the "tweaking" upwards.

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 CloseOnly Mode in Market Can Prevent AMM Withdrawals

Informational | Version 2 | Risk Accepted

CS-BOROS-AMM-016

In Version 2 the new `CLO` *MarketStatus* is introduced, which only allows users to close or reduce their positions. The CloseOnly mode can prevent Liquidity Providers of the AMM from withdrawing their funds: For example, if the LP user currently has 0 size, and the AMM has positive size, the withdrawal would increase the LP user's size through an OTC trade, which would fail because of the `CLO` *MarketStatus*.

Client accepts the risk

Pendle states:

> On closingOnly Mode and AMM withdrawals: it will be part of the AMM LP's terms and conditions (that we will disclaim the users on the frontend also) that their liquidity will be needed when closeOnly mode is turned on and they wont be able to withdraw. In terms of risks, closingOnly mode is turned on when we determine that the market is too heated (either when its near the OI cap, or the market is too volatile vs the liquidity), and hence we do want the AMM liquidity to be there to faciliate potential liquidation

7.2 Initializers for Implementation Contracts Are Not Disabled

Informational | Version 1 | Code Partially Corrected

CS-BOROS-AMM-011

The `initialize` function of `MiscModule` is not disabled in the implementation contract (for example through the constructor).

Likewise, the `initialize` function of the `Router` is not disabled in the implementation contract. `Router` delegatecalls to the `initialize` function of `MiscModule`, but both should be disabled since `Router` will also be used as an implementation contract.

Code partially corrected:

The initializer of `MiscModule` is disabled in the constructor. However, the initializer of `Router` (delegated to `MiscModule`) is not.

7.3 Router Is Unaware of Trade Restrictions

Informational | Version 1 | Acknowledged

CS-BOROS-AMM-012

The AMM enforces restrictions on when it can be traded with. However, the router is not aware of these restrictions and will attempt to route trades through the AMM that revert. For instance, after calculating the optimal route in `calcSwapAmountBookAMM`, the swap will revert in `_swapBookAMM` if:

- the AMM's cutoff timestamp has been reached.
- the AMM has been deleveraged or liquidated.
- the AMM fails the Initial Margin check at the target position.

Acknowledged:

Pendle acknowledges the behavior and chooses to keep it unchanged.

7.4 `setImpliedRateObservationWindow()` Makes the Implied Rate Jump

Informational | Version 1 | Acknowledged

CS-BOROS-AMM-014

The AMM admin calling `setImpliedRateObservationWindow()` makes the oracle implied rate returned by `BaseAMM.oracleImpliedRate()` jump. If the admin changes the window to a shorter value, the oracle implied rate will move towards `lastTradedRate`, while if a longer window is set, the value will jump towards `prevOracleImpliedRate`. The function is not used anywhere in the protocol, however any integrator could be affected by the sporadic behavior.

Acknowledged:

Pendle acknowledges the behavior, but has decided to keep the code unchanged.

8 Notes

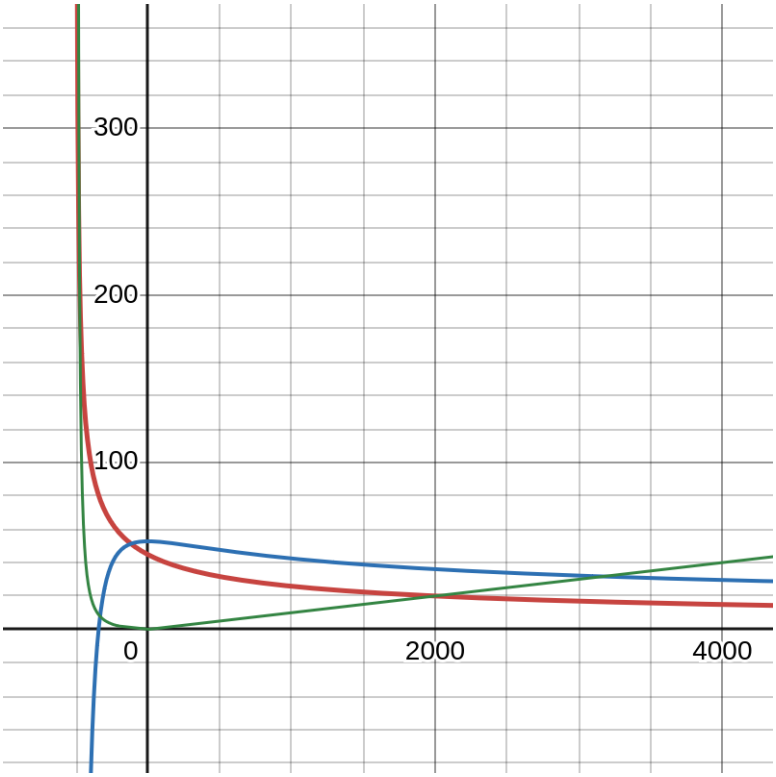
We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 AMM Position Can Fail Initial Margin Check

Note Version 1

Like any other user, the AMM has an account in the `MarketHub`. As such, it must pass the initial margin check whenever it changes its position. The AMM changes its position whenever a trade occurs or when an LP provides or withdraws liquidity. If the initial margin check fails, trading and liquidity provision revert and the AMM must be deleveraged or liquidated. Note that the AMM can approach the initial margin limits if the price moves significantly away from the initial price:

1. If the price continues to decrease, the AMM will accumulate more and more size. Furthermore, the position value decreases due to impermanent loss. The margin requirements increase because size increases by more than the price decreases. At some point, the initial margin requirements are reached.
2. If the price continues to increase, the AMM will accumulate more and more negative size. As in the example above, the position value decreases due to impermanent loss. However, the margin requirements increase much faster because both the size becomes more negative and the price increases. As such, the AMM will not pass the initial margin check for even relative moderate prices.



Trading Invariant (red), Total Value (green), and Total Margin (blue).

In the graph above, the Trading Invariant, Total Value, Total Margin are shown with floating rate tokens on the x-axis and the fixed rate token on the y-axis. Note that portions of the curve where the Total Margin is lower than the Total Value cannot be reached, as the AMM would fail the initial margin check.

$TOTAL\_VALUE >= TOTAL\_MARGIN$

As seen in the graph, even a relatively small increase in the price can lead to a situation where the AMM is unable to pass the initial margin check.

8.2 Agent Can Drain Account Despite Being Limited to Non-destructive Functionality

Note Version 1

The AuthModule of the router limits agents to a selected subset of *TradeModule* and *AMMModule* functionality, that is considered "non-destructive", meaning funds transfers are not allowed. However, the allowed functionality can still be used to extract value from an account, if the agent is compromised. Unfavorable trades can repeatedly be performed by an agent against a malicious counterparty to completely drain an account.

8.3 Behavior of calcSwapSize on Bounds

Note Version 1

The AMM enforces a minimum and maximum price that restricts the trading range. The function `PositiveAMMMath.calcSwapSize` takes a target price and calculates the swap size required to reach that price. However, if the target price lies outside the minimum or maximum bounds (with some rounding threshold), the function will instead use the nearest bound (minimum or maximum price) to calculate the swap size.

The mechanism ensures that users cannot swap tokens at prices outside the allowed range. Integrators should note that if the target price is beyond the bounds, the AMM will not necessarily reach the exact target price after the specified swap is performed.

8.4 Binary Search Upper Bound Possibly Overestimated if Flip Liquidity Initialized to 0

Note Version 1

Binary search is used as the last step of `approxSwapToAddLiquidity()`, in `_calcFinalLiquidity`.

The initial parameters of the binary search are provided by `_getBinSearchParams()`. In case the sweep terminated with `STAGE_SWEPT_ALL`, the `else` branch is taken at line 138 of `LiquidityMath.sol`. Sweeping terminating with `STAGE_SWEPT_ALL` means there is nothing left to trade in the order book, so only the amm will be used beyond the `lastTick` of the sweep. In that case, `guessMax` is initialized as `ammSize.abs() - withAmm.abs()`, which means that it is assumed that the whole `ammSize` can be traded. However, this might not be the case: trading (buying) too much size might move the amm rate above `maxAbsRate`, causing a revert. This revert possibly prevents the binary search from succeeding. To guarantee the binary search will not fail because of `maxAbsRate`, `flipLiquidity` should be initialized such that the flip rate is below `maxAbsRate`.

8.5 Effect of Governance Actions on AMM State

Note Version 1

The AMM administrator can modify several parameters that directly impact the AMM state. Trading on the AMM can be affected as follows:

- If the current fee rate or minimum / maximum rate restriction prevent the AMM from reaching the market price, decreasing the fee or widening the bounds of the AMM allows a trader to backrun the parameter change and move the price closer to the market price.
- If the mimimum / maximum rate are changed so that the current market price is outside the bounds, trades that keep a price outside the bounds revert. However, large trades that push the price back into the bounds succeed.

8.6 Fee Handling in Slippage Protection

Note **Version 1**

In functions that swap with the AMM, the AMM fee is included as part of the cost for the purpose of slippage protection, however the OTC fee is not accounted for in slippage protection.

8.7 Trust Model of Subaccounts

Note **Version 4**

Users interacting with the Boros system via the `AuthModule` rely on relayers to call the router on their behalf. The level of dependency of the user on the relayer varies depending on the type of account used:

- **Main accounts** (`subaccountId 0`) can alternatively be accessed directly by calling the `TradeModule`, `AMMModule`, or `MiscModule`.
- **Subaccounts** (`subaccountId` different from 0) can only be accessed and controlled via relayer. If the relayer shuts down, users lose access to their funds.