Pendle Boros Market & MarketHub Audit Report

Jul 29, 2025

Table of Contents

Summary 2

Overview 3

Issues 4

[WP-I1] Orders fully or mostly handled by AMM do not trigger `_updateImpliedRate()` or use an overall rate that is not from this transaction 4

[WP-M2] `floatingRate` Is Not Multiplied by `timePassed` but Added Directly to `oldIndex.floatingIndex()` 8

[WP-I3] `updateFloatingRate()` should directly use `newFloatingIndex` / `desiredFloatingIndex` instead of `floatingIndexDelta` to prevent unexpected `newIndex.floatingIndex()` caused by unintended `oldIndex.floatingIndex()` . 13

[WP-L4] `setPersonalMarginConfig()` Personal initial margin factors $im_u^{personal}(t)$ less than 1.0 may be detrimental to system health 17

[WP-I5] Regarding The Cost of Manipulating `_updateImpliedRate` 19

[WP-I6] A public cancellation mechanism for risky pending limit orders can reduce the risk of bad debt. 23

[WP-I7] Lack of margin checks for the makers when matching limit orders can cause bad debt. 24

[WP-I8] Consider improving observability of the PendleAccessController role policy 25

[WP-I9] Centralized Risk: Time locks should be added to configuration functions (set*) in `MarketSetAndView` to prevent incorrect configurations from being applied and immediately harming users' interests. 27

[WP-I10] When `fIndexOracle` in `MarketConfigStruct initialConfig` is an `FIndexOracle` contract, it's recommended to deploy both contracts in the same message call to prevent address mismatch in `FIndexOracle.constructor()` . 29

[WP-N11] The finalizeVaultWithdrawal function in MarginManager lacks the onlyRouter modifier, which is inconsistent with the documentation 33

[WP-N12] The current implementation does not support tokens with decimals greater than 18 35

[WP-D13] Support of the Native token as cash was mentioned in the documentation but not presented in the implementation 36

Cash transfer **36**

[WP-M14] `PMath.rawDivCeil()` does not round up when $0 < \frac{x}{d} < 1$ 38

[WP-L15] The residual assets from "remainder of division when converting scaled amount to unscaled during withdrawal `scaledAmount % scalingFactor`" accumulate and become locked in the MarketHub contract. 40

[WP-M16] `cancelVaultWithdrawal()` can be reentered when the `token` has hooks 42

Appendix **44**

Disclaimer **45**

Summary

This report has been prepared for Pendle Boros smart contract, to discover issues and vulnerabilities in the source code of their Smart Contract as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Overview

Project Summary

Project Name	**Pendle Boros**
Codebase	**https://github.com/pendle-finance/pendle-core-v3**
Commit	**a905b3788f13edba3bbdb7e49b5594ae51f28b31**
Language	**Solidity**

Audit Summary

Delivery Date	**Jul 29, 2025**
Audit Methodology	**Static Analysis, Manual Review**
Total Isssues	**16**

[WP-I1] Orders fully or mostly handled by AMM do not trigger `_updateImpliedRate()` or use an overall rate that is not from this transaction

Informational

Issue Description

- When `bookMatched.isZero()` , `_updateImpliedRate(lastMatchedTick, market.k_tickStep)` is not called
- AMM OTC rate is not considered

When sudden market changes (changes in fair market price) clear the limit orders near the old price, and there are not enough limit orders near the new price, we assume temporarily most trades will be made against the AMM.

This may lead to:

Stale `impliedRate` (source of `markRate`)

- RateBound (calculated from `markRate`) deviates from actual prices and blocks normal orders
- `_checkRateDeviation` may block normal orders
- Margin calculations using `markRate` may be unreliable

We argue that the trades with AMM should also update the rate.

https://github.com/pendle-finance/pendle-core-v3/blob/ce40ffdd55abcfd3b61348c2cc8e2b36bcf61a21/contracts/core/market/MarketOrderAndOtc.sol#L30-L60

```
30      function orderAndOtc(
31          MarketAcc userAddr,
32          LongShort memory orders,
33          CancelData memory cancels,
34          OTCTrade[] memory OTCs,
35          int256 critHR
36      ) external onlyMarketHub returns (UserResult memory res, OTCResult[] memory
    otcRes) {
```

```
37          _validateOrderAndOtc(userAddr, orders, OTCs);
38
39          // Phase 1: Read market & user
40          MarketMem memory market = _readMarket({checkPause: YES, checkMaturity:
    YES});
41          (UserMem memory user, PayFee userSettle) = _initUser(userAddr, market);
42          res.settle = userSettle;
43
44          // Phase 2: remove orders
45          res.removedIds = _coreRemoveAft(market, user, cancels, false);
46
47          // Phase 3: Place orders & sum up payment
48          _matchOrder(market, user, orders, res);
49          _coreAdd(market, user, orders, res.bookMatched);
50
51          // Phase 4: OTC
52          if (OTCs.length != 0) {
53              otcRes = new OTCResult[](OTCs.length);
54              _otc(user, OTCs, res, otcRes, market, critHR);
55          }
56
57          // Phase 5: Write state
58          (res.isStrictIM, res.finalVM) = _writeUser(user, market, res.payment,
    orders, critHR, CLOCheck.YES);
59          _writeMarket(market);
60      }
```

```
62      function _matchOrder(
63          MarketMem memory market,
64          UserMem memory user,
65          LongShort memory orders,
66          UserResult memory res
67      ) internal {
68          (
69              Trade bookMatched,
70              Fill partialFill,
71              MarketAcc partialMaker,
72              int16 lastMatchedTick,
73              int256 lastMatchedRate
74          ) = _bookMatch(market.k_tickStep, market.latestFTag, orders);
75
```

```
76          // check validity of the match
77          if (bookMatched.isZero()) return;
78
79          require(!_hasSelfFilledAfterMatch(user, partialMaker, orders.side,
    lastMatchedTick), Err.MarketSelfSwap());
80          require(_checkRateDeviation(lastMatchedRate, market),
    Err.MarketLastTradedRateTooFar());
81
82          // update oracle & merge new match
83          _updateImpliedRate(lastMatchedTick, market.k_tickStep);
84
85          (res.bookMatched, res.partialMaker) = (bookMatched, partialMaker);
86          res.payment = _mergeNewMatchAft(user, market, bookMatched);
87          emit MarketOrdersFilled(user.addr, bookMatched, res.payment.fee());
88
89          // handle partial fill
90          if (partialFill.isZero()) return;
91
92          if (_squashPartial(res.partialMaker, partialFill, market)) {
93              res.partialMaker = AccountLib.ZERO_MARKET_ACC;
94              return;
95          }
96
97          (UserMem memory partialUser, PayFee partialSettle) =
    _initUser(res.partialMaker, market);
98          res.partialPayFee =
    partialSettle.addPayment(_mergePartialFillAft(partialUser, market, partialFill));
99          _writeUserNoCheck(partialUser, market);
100     }
```

Response from Pendle team:
In practice, Pendle team will run bots that makes small orders on the orderbook to make sure that the filled orders and Implied rate on the orderbook is in sync with the AMM rate;
There will also be active arbitrage bots that will keep the AMM implied rate in sync with the orderbook's rate;
If the AMM makes up an outsized part of the trading volume, the Pendle team will consider taking into account the AMM rate into the Implied Rate in a future upgrade.

Status

ⓘ Acknowledged

[WP-M2] `floatingRate` Is Not Multiplied by `timePassed` but Added Directly to `oldIndex.floatingIndex()`

Medium

Issue Description

Based on how FIndexOracle compares floating rate with `markRate`, we infer that floating rate should be a value comparable to mark rate:

```
49      function updateFloatingRate(int112 floatingRate, uint32 desiredTimestamp)
    external onlyKeeper {
50          FIndexOracleParams memory params = _params;
51
52          int256 markRate = IMarket(params.market).getMarkRate();
53          require((markRate - floatingRate).abs() <= params.maxFRateDeviation,
    Err.FIndexDeviationTooHigh());
54
55          FIndex oldIndex = _latestFIndex;
56          uint32 blockTimestamp = uint32(block.timestamp);
57
58          (uint32 lastUpdateTime, uint32 nextUpdateTimestamp) =
    _calcUpdateTime(params, oldIndex, blockTimestamp);
59          require(lastUpdateTime < params.maturity, Err.FIndexUpdatedAtMaturity());
60          require(nextUpdateTimestamp <= blockTimestamp,
    Err.FIndexNotDueForUpdate());
61          require(desiredTimestamp == nextUpdateTimestamp, Err.FIndexInvalidTime());
62
63          FIndex newIndex = _calcNewFIndex(params, floatingRate, oldIndex,
    nextUpdateTimestamp);
64          _latestFIndex = newIndex;
65          _latestAnnualizedRate = _calcAnnualizedRateBetween(oldIndex, newIndex);
66
67          IMarket(params.market).updateFIndex(newIndex);
68      }
```

However, in the _calcNewFIndex function, floating rate is directly added to the previous floating rate old value to get a new value, which is very different from how feeIndex is calculated:

```
169     function _calcNewFIndex(
170         FIndexOracleParams memory params,
171         int112 floatingRate,
172         FIndex oldIndex,
173         uint32 timestamp
174     ) internal pure returns (FIndex newIndex) {
175         // assert(!oldIndex.isZero());
176         int112 floatingIndex = floatingRate + oldIndex.floatingIndex();
177         uint64 feeIndex = PaymentLib.calcNewFeeIndex(
178             oldIndex.feeIndex(),
179             params.settleFeeRate,
180             timestamp - oldIndex.fTime()
181         );
182 
183         newIndex = FIndexLib.from(timestamp, floatingIndex, feeIndex);
184     }
```

We expect both floatingIndex and feeIndex to be an area calculation (`rate(t) dt`).

Details on calculation of new fee index

```
36     function calcNewFeeIndex(uint64 oldFeeIndex, uint256 feeRate, uint32
timePassed) internal pure returns (uint64) {
37         return oldFeeIndex + calcFloatingFee(PMath.ONE, feeRate,
timePassed).Uint64();
38     }
```

```
11     function calcFloatingFee(uint256 absSize, uint256 feeRate, uint32 timeToMat)
internal pure returns (uint256) {
12         return (absSize * feeRate * timeToMat).rawDivUp(PMath.ONE_MUL_YEAR);
13     }
```

```
631     function from(uint32 _fTime, int112 _floatingIndex, uint64 _feeIndex) internal
pure returns (FIndex) {
632         uint208 rawValue = (uint208(_fTime) << 176) |
(uint208(uint112(_floatingIndex)) << 64) | uint208(_feeIndex);
633         return FIndex.wrap(bytes26(rawValue));
634     }
```

As we can see, when using floating index, it follows the same pattern as fee index by multiplying the difference with size to calculate settlement amounts:

```
98      function __processSweptUntilStop(
99          UserMem memory user,
100         MarketMem memory market,
101         SweptF[] memory longSweptF,
102         SweptF[] memory shortSweptF,
103         FTag stopFTag,
104         Trade tradeAtStop
105     ) private view returns (PayFee res) {
106         FIndex userIndex = _toFIndex(user.fTag);
107         do {
108             FTag thisTag = stopFTag;
109             if (longSweptF.length > 0) thisTag =
    thisTag.min(longSweptF[longSweptF.length.dec()].fTag);
110             if (shortSweptF.length > 0) thisTag =
    thisTag.min(shortSweptF[shortSweptF.length.dec()].fTag);
111
112             FIndex thisIndex = _toFIndex(thisTag);
113
114             res = res + Pay.calcSettlement(user.signedSize, userIndex, thisIndex);
115             (user.fTag, userIndex) = (thisTag, thisIndex);
116
117             Trade sumTrade = thisTag == stopFTag ? tradeAtStop : TradeLib.ZERO;
118             sumTrade =
119                 sumTrade +
120                 __iterateSweptSameTag(thisTag, longSweptF) +
121                 __iterateSweptSameTag(thisTag, shortSweptF);
122
123             if (sumTrade.isZero()) continue;
124             user.signedSize += sumTrade.signedSize();
125
126             int256 upfrontCost = sumTrade.toUpfrontFixedCost(market.k_maturity -
    userIndex.fTime());
127             res = res.subPayment(upfrontCost);
128         } while (user.fTag != stopFTag);
129     }
```

```
15      function calcSettlement(int256 signedSize, FIndex last, FIndex current)
    internal pure returns (PayFee res) {
```

```
16          if (last == current) return PLib.ZERO;
17
18          res = PLib.from(
19              signedSize.mulFloor(current.floatingIndex() - last.floatingIndex()),
20              signedSize.abs().mulUp(current.feeIndex() - last.feeIndex())
21          );
22      }
```

Details on settlements and payments

```
55      function orderAndOtc(
56          MarketId marketId,
57          MarketAcc mainUser,
58          LongShort memory orders,
59          CancelData memory cancelData,
60          OTCTrade[] memory OTCs
61      ) external onlyRouter returns (Trade /*bookMatched*/, uint256
    /*totalTakerOtcFee*/) {
62          bool checkCritHealthUser = _checkEnteredMarketsAndStrictHealth(mainUser,
    marketId);
63          bool[] memory checkCritHealthOTCs = new bool[](OTCs.length);
64          for (uint256 i = 0; i < OTCs.length; i++) {
65              checkCritHealthOTCs[i] =
    _checkEnteredMarketsAndStrictHealth(OTCs[i].counter, marketId);
66          }
67
68          address market = _marketIdToAddrRaw(marketId);
69          (UserResult memory userRes, OTCResult[] memory otcRes) =
    IMarket(market).orderAndOtc(
70              mainUser,
71              orders,
72              cancelData,
73              OTCs,
74              critHR
75          );
76
77          _processPayFee(mainUser, userRes.settle + userRes.payment);
78          if (!userRes.partialMaker.isZero()) {
79              _processPayFee(userRes.partialMaker, userRes.partialPayFee);
80          }
81          for (uint256 i = 0; i < OTCs.length; i++) {
```

```
82              _processPayFee(OTCs[i].counter, otcRes[i].settle + otcRes[i].payment);
83          }
84
85          _processMarginCheck(mainUser, marketId, userRes.isStrictIM,
    checkCritHealthUser, userRes.finalVM);
86          for (uint256 i = 0; i < OTCs.length; i++) {
87              _processMarginCheck(
88                  OTCs[i].counter,
89                  marketId,
90                  otcRes[i].isStrictIM,
91                  checkCritHealthOTCs[i],
92                  otcRes[i].finalVM
93              );
94          }
95
96          return (userRes.bookMatched, userRes.payment.fee());
97      }
```

```
183     function _processPayFee(MarketAcc user, PayFee payFee) internal {
184         (int128 payment, uint128 fees) = payFee.unpack();
185         if (payment != 0 || fees != 0) {
186             acc[user].cash += payment - uint256(fees).Int();
187         }
188         if (fees != 0) {
189             cashFeeData[user.tokenId()].treasuryCash += fees;
190         }
191     }
```

Status

✓ Fixed

[WP-I3] `updateFloatingRate()` should directly use `newFloatingIndex` / `desiredFloatingIndex` instead of `floatingIndexDelta` to prevent unexpected `newIndex.floatingIndex()` caused by unintended `oldIndex.floatingIndex()` .

Informational

Issue Description

By switching to use newFloatingIndex / desiredFloatingIndex, when updating the floating index for epoch n, we don't need to care about whether the updates for epoch 1 through epoch n-1 were successful.

To simplify the description:

- Let `params.maturity - a * params.updatePeriod` be denoted as $epochTime_a$
- Let `(params.maturity - a * params.updatePeriod) + n * params.updatePeriod` be denoted as $epochTime_{a+n}$

Given:

- When executing `updateFloatingRate()` at a time where `block.timestamp - params.maxUpdateDelay` is within $[epochTime_a, epochTime_{a+1})$ (inclusive, exclusive): `_latestFIndex` updates to:
  { fTime: $epochTime_{a+1}$, floatingIndex: startIndex }

Normal scenario:

- When executing updateFloatingRate($floatingIndexDelta_1$, $epochTime_{a+2}$) at a time where `block.timestamp - params.maxUpdateDelay` is within $[epochTime_{a+1}, epochTime_{a+2})$: `_latestFIndex` updates to:
  { fTime: $epochTime_{a+2}$, floatingIndex: $startIndex + floatingIndexDelta_1$ }
- When executing updateFloatingRate($floatingIndexDelta_2$, $epochTime_{a+3}$) at a time where `block.timestamp - params.maxUpdateDelay` is within $[epochTime_{a+2}, epochTime_{a+3})$: `_latestFIndex` updates to:

{ fTime: $epochTime_{a+3}$, floatingIndex:
$startIndex + floatingIndexDelta_1 + floatingIndexDelta_2$ }

Edge case:

- The transaction containing updateFloatingRate($floatingIndexDelta_1$, $epochTime_{a+2}$) failed or reverted (due to off-chain reasons like bot run out of balance or on-chain reasons like network congestion)
- When executing updateFloatingRate($floatingIndexDelta_2$, $epochTime_{a+3}$) at a time where `block.timestamp - params.maxUpdateDelay` is within $[epochTime_{a+2}, epochTime_{a+3})$:
  `_latestFIndex` updates to:
  { fTime: $epochTime_{a+3}$, floatingIndex: $startIndex + floatingIndexDelta_2$ }
  `floatingIndex` misses $floatingIndexDelta_1$

To prevent such edge cases, constructing each `updateFloatingRate()` calldata requires accurate observation of the on-chain `floatingIndex` state.

In fact, the original business requirement is to update the `floatingIndex` at `desiredTimestamp` to a desired floating index value. Using the absolute desired floating index value instead of delta removes dependency on the success of previous `updateFloatingRate()` transactions.

Besides being more accurate and predictable, constructing `updateFloatingRate()` calldata by directly using the expected final state value at `desiredTimestamp`, rather than relying on delta values from the on-chain state, may also be simpler and more straightforward.

```
49      function updateFloatingRate(int112 floatingRate, uint32 desiredTimestamp)
    external onlyKeeper {
50          FIndexOracleParams memory params = _params;
51
52          int256 markRate = IMarket(params.market).getMarkRate();
53          require((markRate - floatingRate).abs() <= params.maxFRateDeviation,
    Err.FIndexDeviationTooHigh());
54
55          FIndex oldIndex = _latestFIndex;
56          uint32 blockTimestamp = uint32(block.timestamp);
57
58          (uint32 lastUpdateTime, uint32 nextUpdateTimestamp) =
    _calcUpdateTime(params, oldIndex, blockTimestamp);
59          require(lastUpdateTime < params.maturity, Err.FIndexUpdatedAtMaturity());
60          require(nextUpdateTimestamp <= blockTimestamp,
    Err.FIndexNotDueForUpdate());
```

```
61          require(desiredTimestamp == nextUpdateTimestamp, Err.FIndexInvalidTime());
62
63          FIndex newIndex = _calcNewFIndex(params, floatingRate, oldIndex,
    nextUpdateTimestamp);
64          _latestFIndex = newIndex;
65          _latestAnnualizedRate = _calcAnnualizedRateBetween(oldIndex, newIndex);
66
67          IMarket(params.market).updateFIndex(newIndex);
68      }
```

```
151     function _calcUpdateTime(
152         FIndexOracleParams memory params,
153         FIndex latestIndex,
154         uint32 blockTimestamp
155     ) internal pure returns (uint32 lastUpdateTime, uint32 nextUpdateTimestamp) {
156         lastUpdateTime = latestIndex.fTime();
157
158         // nextUpdateTimestamp:
159         // - must be an epoch endpoint.
160         // - must not be before lastUpdateTime
161         // - must not be before block.timestamp - maxDelay
162         //   (in other words, block.timestamp <= nextUpdateTimestamp + maxDelay)
163         (, nextUpdateTimestamp) = _calcEpochRange(
164             params,
165             _max(lastUpdateTime, _subMax0(blockTimestamp, params.maxUpdateDelay))
166         );
167     }
```

```
130     /// epoch endpoints have the form of (maturity - k * period)
131     function _calcEpochRange(
132         FIndexOracleParams memory params,
133         uint32 timestamp
134     ) internal pure returns (uint32 start, uint32 stop) {
135         uint32 maturity_ = params.maturity;
136         uint32 period = params.updatePeriod;
137         if (timestamp >= maturity_) return (maturity_, maturity_);
138
139         // find smallest k such that
140         //      (maturity - k * period) <= timestamp
```

```
141         //      maturity - timestamp <= k * period
142         //
143         //      k = ceil((maturity - timestamp) / period)
144
145         uint32 k = (maturity_ - timestamp + period - 1) / period;
146         // slither-disable-next-line divide-before-multiply
147         start = params.maturity - k * period;
148         stop = start + period;
149     }
```

```
169     function _calcNewFIndex(
170         FIndexOracleParams memory params,
171         int112 floatingRate,
172         FIndex oldIndex,
173         uint32 timestamp
174     ) internal pure returns (FIndex newIndex) {
175         // assert(!oldIndex.isZero());
176         int112 floatingIndex = floatingRate + oldIndex.floatingIndex();
177         uint64 feeIndex = PaymentLib.calcNewFeeIndex(
178             oldIndex.feeIndex(),
179             params.settleFeeRate,
180             timestamp - oldIndex.fTime()
181         );
182
183         newIndex = FIndexLib.from(timestamp, floatingIndex, feeIndex);
184     }
```

Response from Pendle team:
In our operation, we will carefully check that the oldIndex is consistent (and hence the new Floating Index will be the expected one) before updating with floatingIndexDelta.

Status

ⓘ Acknowledged

[WP-L4] `setPersonalMarginConfig()` Personal initial margin factors $im_u^{personal}(t)$ less than 1.0 may be detrimental to system health

Low

Issue Description

Current implementation of `setPersonalMarginConfig()` has no restrictions on `newKIM` and `newKMM` .

https://github.com/pendle-finance/pendle-core-v3/blob/4f769483ded7dff9afd226bba9e0e171c8e95a26/contracts/core/market/MarketSetAndView.sol#L73-L76

```
73      function setPersonalMarginConfig(MarketAcc user, uint64 newKIM, uint64 newKMM)
    external onlyAuthorized {
74          (_accState(user).kIM, _accState(user).kMM) = (newKIM, newKMM);
75          emit PersonalMarginConfigUpdated(user, newKIM, newKMM);
76      }
```

```
60      function _kIM(MarketAcc user) internal view returns (uint64) {
61          uint64 kIMAcc = _accState(user).kIM;
62          return kIMAcc != 0 ? kIMAcc : _ctx().kIM;
63      }
64
65      function _kMM(MarketAcc user) internal view returns (uint64) {
66          uint64 kMMAcc = _accState(user).kMM;
67          return kMMAcc != 0 ? kMMAcc : _ctx().kMM;
68      }
```

```
27      /// @dev Aft suffix means the call must only be post-processing the user
28      function _getIMAft(MarketMem memory market, UserMem memory user) internal view
    returns (VMResult) {
29          return VMResultLib.from(_getValueAft(market, user), _calcIM(market, user,
    _kIM(user.addr)));
```

```
30      }
31
32      function _getMMAft(MarketMem memory market, UserMem memory user) internal view
    returns (VMResult) {
33          return VMResultLib.from(_getValueAft(market, user), _calcMM(market,
    user.signedSize, _kMM(user.addr)));
34      }
35
36      function _calcIM(MarketMem memory market, UserMem memory user, uint64 kIM)
    internal pure returns (uint256) {
37          uint256 PM = _calcPM(market, user);
38          return (PM * kIM *
    market.timeToMat.max(market.tThresh)).rawDivUp(PMath.ONE_MUL_YEAR);
39      }
40
41      /// @dev long, complicated formulas following the white-paper strictly
42      function _calcMM(MarketMem memory market, int256 signedSize, uint64 kMM)
    internal pure returns (uint256) {
43          uint256 absSize = signedSize.abs();
44          uint256 absMarkRate = market.rMark.abs();
45
46          if (absMarkRate > market.k_iThresh) {
47              uint256 scaledTThresh = market.tThresh.mulUp(kMM);
48              if (market.timeToMat < scaledTThresh && signedSize * market.rMark > 0)
    {
49                  return
50                      (absSize * (absMarkRate * market.timeToMat + market.k_iThresh
    * (scaledTThresh - market.timeToMat)))
51                          .rawDivUp(PMath.ONE_MUL_YEAR);
52              }
53              // else use the calculation outside
54          }
55
56          uint256 PM = __calcPMFromRate(absSize, market.rMark, market.k_iThresh);
57          return (PM * kMM *
    market.timeToMat.max(market.tThresh)).rawDivUp(PMath.ONE_MUL_YEAR);
58      }
```

[WP-I5] Regarding The Cost of Manipulating `_updateImpliedRate`

Informational

Issue Description

```
22  function _updateImpliedRate(int16 lastMatchedTick, uint8 tickStep) internal {
23      uint32 blockTimestamp = uint32(block.timestamp);
24      _ctx().impliedRate = _ctx().impliedRate.update(blockTimestamp,
    lastMatchedTick, tickStep);
25  }
```

When the latest trade occurs outside the window, its price is directly used as the `OracleImpliedRate` , regardless of the transaction size.

For a market with a sparse order book, an attacker may observe a recent out-of-window timing and make a trade against the order book to update the implied rate at a relatively low cost. This can be further optimized by making an opposite trade against the AMM after the price manipulation on the order book.

This might be exploited by prematurely liquidating accounts near the liquidation line.

As the oracle price will be used for liquidation, it must be a price that can resist manipulation, which is usually achieved by waiting for sudden price changes to be adjusted back to their fair market price by arbitrage trading. A proper TWAP is needed in order to achieve such an effect.

```
62  function _matchOrder(
63          MarketMem memory market,
64          UserMem memory user,
65          LongShort memory orders,
66          UserResult memory res
67      ) internal {
68          (
69              Trade bookMatched,
70              Fill partialFill,
71              MarketAcc partialMaker,
```

```
72              int16 lastMatchedTick,
73              int256 lastMatchedRate
74          ) = _bookMatch(market.k_tickStep, market.latestFTag, orders);
75
76          // check validity of the match
77          if (bookMatched.isZero()) return;
78
79          require(!_hasSelfFilledAfterMatch(user, partialMaker, orders.side,
    lastMatchedTick), Err.MarketSelfSwap());
80          require(_checkRateDeviation(lastMatchedRate, market),
    Err.MarketLastTradedRateTooFar());
81
82          // update oracle & merge new match
83          _updateImpliedRate(lastMatchedTick, market.k_tickStep);
84
85          (res.bookMatched, res.partialMaker) = (bookMatched, partialMaker);
86          res.payment = _mergeNewMatchAft(user, market, bookMatched);
87          emit MarketOrdersFilled(user.addr, bookMatched, res.payment.fee());
88
89          // handle partial fill
90          if (partialFill.isZero()) return;
91
92          if (_squashPartial(res.partialMaker, partialFill, market)) {
93              res.partialMaker = AccountLib.ZERO_MARKET_ACC;
94              return;
95          }
96
97          (UserMem memory partialUser, PayFee partialSettle) =
    _initUser(res.partialMaker, market);
98          res.partialPayFee =
    partialSettle.addPayment(_mergePartialFillAft(partialUser, market, partialFill));
99          _writeUserNoCheck(partialUser, market);
100     }
101
```

```
77  function _calcOracleImpliedRate(
78          MarketImpliedRate self,
79          uint32 blockTimestamp,
80          uint8 tickStep
81      )
82          internal
```

```
        pure
        returns (
            // unpacked data
            uint32 _window,
            uint32 _lastTradedTime,
            // calculated data
            int128 _lastTradedRate,
            int128 oracleRate
        )
    {
        int128 _prevOracleRate;
        int16 _lastTradedTick;
        (_prevOracleRate, _window, _lastTradedTime, _lastTradedTick) = self.unpack();
        _lastTradedRate = TickMath.getRateAtTick(_lastTradedTick, tickStep);
        oracleRate = FixedWindowObservationLib.calcCurrentOracleRate(
            _prevOracleRate,
            _window,
            _lastTradedTime,
            _lastTradedRate,
            blockTimestamp
        );
    }
```

```
function calcCurrentOracleRate(
        int128 prevOracleRate,
        uint32 window,
        uint32 lastTradedTime,
        int128 lastTradedRate,
        uint32 blockTimestamp
    ) internal pure returns (int128) {
        int256 iWindow = int256(uint256(window));
        int256 timeElapsed = int32(blockTimestamp - lastTradedTime);
        if (timeElapsed == 0) return prevOracleRate;
        if (timeElapsed >= iWindow) {
            return lastTradedRate;
        }
        int256 oracleRate = int256(lastTradedRate) * timeElapsed + int256(prevOracleRate) * (iWindow - timeElapsed);
        oracleRate /= iWindow;
        return oracleRate.Int128();
```

```
34         }
```

https://docs.notional.finance/governance/fcash-valuation/interest-rate-oracles

Status

ⓘ **Acknowledged**

[WP-I6] A public cancellation mechanism for risky pending limit orders can reduce the risk of bad debt.

Informational

Issue Description

In the current implementation, pending limit orders are considered in IM calculations.

However, two additional factors that can passively impact the user's account health ratio are not considered:

1. The change in the value of existing positions from the current price (at the time of placing the limit order) to the pending limit order price (change of $p_{unrealized}(t)$);
2. The dynamic settlements for existing positions (change of `cash`).

Consequently, when the limit order gets filled, the maker's account can immediately become liquidatable, and in the worst-case scenario, it may even cause bad debt to the system.

Relying solely on margin checks during order matching would be gas intensive. We suggest implementing a new mechanism to cancel limit orders near the current price that are deemed risky.

Similar to liquidations, we propose a public mechanism (potentially with rewards) where anyone can cancel pending limit orders of users with insufficient margin (this could be checked to determine whether the limit order is still allowed to be placed based on the current oracle price; if not, then it can be canceled).

> `_bookPurgeOob` (in OrderBookUtils.sol) is a similar protection mechanism, but it doesn't consider the order maker's account states. A global bound that is too strict will constrain the flexibility to place safe limit orders for accounts with enough margin.

Status

Acknowledged

[WP-I7] Lack of margin checks for the makers when matching limit orders can cause bad debt.

Informational

Issue Description

As we mentioned in **[WP-I6] A public cancellation mechanism for risky pending limit orders can reduce the risk of bad debt**, the IM check at the time the limit order is placed is not enough to ensure the account is healthy when the limit order is filled.

The current flow of order book matching:

- A taker calls `orderAndOtc` (`router` -> `MarketHub.orderAndOtc` -> `Market.orderAndOtc`);
- Matching happens inside `_bookMatch` and produces `partialMaker` (the passive account hit);
- For the taker, we again call `_writeUser` → `_checkMargin` ;
- For the **passive maker**, we do **not** call `_writeUser` ;

We believe canceling these orders or reverting when the maker's margin check fails would be better for keeping the system healthy than allowing the fulfillment of such orders.

Status

ⓘ Acknowledged

[WP-I8] Consider improving observability of the PendleAccessController role policy

Informational

Issue Description

`PendleAccessController` governs access control for many critical system functions.

The current implementation stores (target, selector) -> role configuration in the `mapping(address => mapping(bytes4 => bytes32)) public allowedRoles` mapping, which is not enumerable.

Additionally, `setAllowedRole()` emits no events. This makes it difficult to verify the current `PendleAccessController` configuration.

Consider adding events for `setAllowedRole()`, and using a data structure like `abi.encodePacked(target, selector)` -> role `EnumerableMap.Bytes32ToBytes32Map` along with functions to inspect the current configuration.

```
11  contract PendleAccessController is
12      IPAccessManagerCore,
13      AccessControlEnumerableUpgradeable,
14      PendleRolesConstants,
15      UUPSUpgradeable
16  {
17      mapping(address => mapping(bytes4 => bytes32)) public allowedRoles;
18
    @@ 19,27 @@
28
29      function _authorizeUpgrade(address) internal view override
    onlyRole(DEFAULT_ADMIN_ROLE) {}
30
31      function setAllowedRole(address target, bytes4 selector, bytes32 role)
    external onlyRole(DEFAULT_ADMIN_ROLE) {
32          allowedRoles[target][selector] = role;
33      }
34
35      function canCall(address caller, address target, bytes4 selector) external
    view returns (bool) {
```

```
36          return hasRole(allowedRoles[target][selector], caller) ||
    hasRole(DEFAULT_ADMIN_ROLE, caller);
37      }
38
    @@ 39,41 @@
42  }
```

Status

ⓘ Acknowledged

[WP-I9] Centralized Risk: Time locks should be added to configuration functions (set*) in `MarketSetAndView` to prevent incorrect configurations from being applied and immediately harming users' interests.

Informational

Issue Description

Potential malicious actions through configuration changes:

1. Malicious Liquidation:
   - The `setGlobalMarginConfig` function can maliciously increase the initial margin (`newKIM`) and maintenance margin (`newKMM`).
   - Through the `setGlobalOracleAddresses` function, the oracle (`newMarkRateOracle`) can be replaced with a malicious one controlled by the attacker to report extreme prices.
   - Through the `setGlobalLiquidationSettings` function, the liquidation fee (`liqSettings.feeRate`) can be set to 100%.
   - Any of these operations would instantly make normal user positions liquidatable, allowing attackers to immediately liquidate these positions.
2. Trading Fee Theft:
   - Through the `setGlobalFeeRates` function, the taker fee (`newTakerFee`) can be set to an extreme value, such as 100%.
   - Without users' knowledge, their next market order would result in most of their funds being seized as fees by the protocol.
3. The ability to modify `PersonalMarginConfig` at any time may lead to unexpected liquidation due to increased collateral requirements that users are unaware of (including immediate liquidation when configuration is modified)
   - When admin sets someone's health requirements lower than global requirements, admin-related personnel gain unfair advantages
   - When admin sets someone's health requirements higher than global requirements, regular users face hidden liquidation risks

   If this functionality is truly necessary, perhaps only stricter individual requirements should be allowed, or the global config could be used when personal config is more

lenient.

Status

ⓘ Acknowledged

[WP-I10] When `fIndexOracle` in `MarketConfigStruct initialConfig` is an `FIndexOracle` contract, it's recommended to deploy both contracts in the same message call to prevent address mismatch in `FIndexOracle.constructor()` .

Informational

Issue Description

In the current implementation, if a new market needs `fIndexOracle` that is an `FIndexOracle` contract, the `FIndexOracle` contract should be deployed first, and its address should be passed in the parameters of the `MarketFactory.create()` call.

- `MarketFactory.create()` requires `fIndexOracle` and calls `fIndexOracle.getLatestFIndex()`
- `FIndexOracle.constructor()` requires the market address, and `FIndexOracle` has no interface to modify the market address after deployment.

However, to deploy an `FIndexOracle` contract, the market's address is needed in its `constructor` .

Therefore, when executing `FIndexOracle.constructor()` , an accurate prediction of the market's address before actual deployment is required.

If other `MarketFactory.create()` calls occur between the execution of `FIndexOracle.constructor()` and `MarketFactory.create()` (which includes the `fIndexOracle` address parameter from the former), the predicted market address used in `FIndexOracle.constructor()` will not match the actual address.

It might be more robust to do `new FIndexOracle(...)` within `MarketFactory.create()` for such cases.

```
43      function create(
44          string memory name,
45          string memory symbol,
46          bool isIsolatedOnly,
47          uint32 maturity,
48          TokenId tokenId,
```

```
49          uint8 tickStep,
50          uint16 iTickThresh,
51          MarketConfigStruct memory config,
52          MarketImpliedRateLib.InitStruct memory impliedRateInit
53      ) external onlyAuthorized returns (address newMarket) {
54          require(IMarketHub(MARKET_HUB).tokenData(tokenId).token != address(0),
    Err.InvalidTokenId());
55
56          MarketId newMarketId = MarketId.wrap(++marketNonce);
57          MarketImmutableDataStruct memory immData = MarketImmutableDataStruct({
58              name: name,
59              symbol: symbol,
60              k_isIsolatedOnly: isIsolatedOnly,
61              k_maturity: maturity,
62              k_tokenId: tokenId,
63              k_marketId: newMarketId,
64              k_tickStep: tickStep,
65              k_iTickThresh: iTickThresh
66          });
67
68          newMarket = address(
69              new TransparentUpgradeableProxy(
70                  IMPLEMENTATION,
71                  msg.sender,
72                  abi.encodeCall(IMarketSetting.initialize, (immData, config,
    impliedRateInit))
73              )
74          );
75
76          address computedAddress = CreateCompute.compute(address(this),
    newMarketId);
77          assert(computedAddress == newMarket);
78
79          assert(allMarkets.add(newMarket));
80
81          emit MarketCreated(newMarket, immData, config);
82      }
```

```
34      function initialize(
35          MarketImmutableDataStruct memory initialImmData,
36          MarketConfigStruct memory initialConfig,
```

```
37          MarketImpliedRateLib.InitStruct memory impliedRateInit
38      ) external initializer onlyRole(_INITIALIZER_ROLE) {
39          _setImmData(initialImmData);
40
41          _setGlobalMaxOpenOrders(initialConfig.maxOpenOrders);
42          _setGlobalOracleAddresses(initialConfig.markRateOracle,
    initialConfig.fIndexOracle);
    @@ 43,59 @@
60
61
    _updateFIndex(IFIndexOracle(initialConfig.fIndexOracle).getLatestFIndex());
62          _initImpliedRateOracle(impliedRateInit);
63      }
```

```
12  contract FIndexOracle is IFIndexOracle, PendleRolesPlugin {
13      using PMath for int256;
14
15      struct FIndexOracleParams {
16          // fixed param
17          uint32 maturity;
18          address market;
19          // config
20          uint64 settleFeeRate;
21          uint32 updatePeriod;
22          uint32 maxUpdateDelay;
23          uint128 maxFRateDeviation;
24      }
25
26      FIndexOracleParams internal _params;
    @@ 27,34 @@
35
36      constructor(
37          FIndexOracleParams memory params,
38          address permissionController_,
39          address keeper_
40      ) PendleRolesPlugin(permissionController_) {
41          _params = params;
42
```

```
@@ 43,46 @@
47      }
48
@@ 49,201 @@
202 }
```

Status

ⓘ Acknowledged

[WP-N11] The finalizeVaultWithdrawal function in MarginManager lacks the onlyRouter modifier, which is inconsistent with the documentation

Issue Description

https://github.com/pendle-finance/pendle-core-v3/blob/619109de20fcefe421765719438473807bb0513f/documentation/codebase/markethub/MarketHub.md#L1-L9

```
1  # MarketHub
2
3  _MarketHub_ is where all the markets are managed. _MarketHub_ is the main entry
   point for the user to interact with the system, including cash transfer and
   _Market_ interaction. _MarketHub_ is also used to maintain the system cross
   invariant.
4
5  ## System interaction
6
7  All _Markets_'s (non-view) operations can only be called by _MarketHub_.
8
9  _MarketHub_'s (non-view) operations can only be called by _Router_.
```

```
89      /// @dev allow this to be called by anyone
90      function finalizeVaultWithdrawal(address root, TokenId tokenId) external {
91          TokenData memory data = tokenData(tokenId);
92          Withdrawal memory user = _withdrawal[root][tokenId];
93
94          require(uint256(user.start) + uint256(getPersonalCooldown(root)) <=
    block.timestamp, Err.MHWithdrawNotReady());
95
96          IERC20(data.token).safeTransfer(root, user.unscaled);
97          emit VaultWithdrawalFinalized(root, tokenId, user.unscaled);
98
99          user.unscaled = 0;
100         _withdrawal[root][tokenId] = user;
101     }
```

Status

✓ Fixed

[WP-N12] The current implementation does not support tokens with decimals greater than 18

Issue Description

And also tokens without the `decimals()` function.

https://github.com/pendle-finance/pendle-core-v3/blob/4f769483ded7dff9afd226bba9e0e171c8e95a26/contracts/core/markethub/Storage.sol#L202

```
191     /// @dev in setting up this setMarketEntranceFees & setMinCashForAccounts must be called too
192     function registerToken(address token) external onlyAuthorized returns (TokenId newTokenId) {
193         uint256 len = _tokenData.length;
194         for (uint256 i = 0; i < len; i++) {
195             require(_tokenData[i].token != token, Err.MHTokenExists());
196         }
197         require(len < type(uint16).max, Err.MHTokenLimitExceeded());
198 
199         newTokenId = TokenId.wrap(uint16(_tokenData.length));
200 
201         // @dev safe cast scaling factor <= 10**18
202         uint96 scalingFactor = uint96(10 ** (18 - IERC20Metadata(token).decimals()));
203         _tokenData.push(TokenData(token, scalingFactor));
204 
205         emit TokenAdded(newTokenId, token);
206     }
```

Status

ⓘ Acknowledged

[WP-D13] Support of the Native token as cash was mentioned in the documentation but not presented in the implementation

Issue Description

https://github.com/pendle-finance/pendle-core-v3/blob/619109de20fcefe421765719438473807bb0513f/documentation/codebase/markethub/MarketHub.md#L11-L13

> # Cash transfer
> Users can deposit and withdraw cash in and out of the system. Cash can either be of the ==native token== or an ERC20 token. All cash deposited into the system is held by the *MarketHub* contract.

```
54      function vaultDeposit(MarketAcc acc, uint256 unscaled) external onlyRouter {
55          TokenData memory data = tokenData(acc.tokenId());
56
57          IERC20(data.token).safeTransferFrom(msg.sender, address(this), unscaled);
58          _topUpWithdrawCash(acc, _toScaled(unscaled, data.scalingFactor), true);
59
60          emit VaultDeposit(acc, unscaled);
61      }
62

@@ 63,87 @@

88
89      /// @dev allow this to be called by anyone
90      function finalizeVaultWithdrawal(address root, TokenId tokenId) external {
91          TokenData memory data = tokenData(tokenId);
92          Withdrawal memory user = _withdrawal[root][tokenId];
93
94          require(uint256(user.start) + uint256(getPersonalCooldown(root)) <=
block.timestamp, Err.MHWithdrawNotReady());
95
96          IERC20(data.token).safeTransfer(root, user.unscaled);
97          emit VaultWithdrawalFinalized(root, tokenId, user.unscaled);
98
99          user.unscaled = 0;
```

```
100             _withdrawal[root][tokenId] = user;
101         }
```

Status

✓ Fixed

[WP-M14] `PMath.rawDivCeil()` does not round up when $0 < \frac{x}{d} < 1$

Medium

Issue Description

For example, this test

```
11      function test_rawDivCeil() public {
12          assertEq(PMath.rawDivCeil(11, 10), 2);
13          assertEq(PMath.rawDivCeil(10, 10), 1);
14          assertEq(PMath.rawDivCeil(9, 10), 1);
15      }
```

fails at L14:

```
[4911] PMathTest::test_rawDivCeil()
  ├─ [0] VM::assertEq(2, 2) [staticcall]
  │   └─ ← [Return]
  ├─ [0] VM::assertEq(1, 1) [staticcall]
  │   └─ ← [Return]
  ├─ [0] VM::assertEq(0, 1) [staticcall]
  │   └─ ← [Revert] assertion failed: 0 != 1
  └─ ← [Revert] assertion failed: 0 != 1
```

The root cause of this issue is that the sign used to determine whether to round up or down is from the result of native `div`, which always rounds down towards zero:

```
111     function rawDivCeil(int256 x, int256 d) internal pure returns (int256 z) {
112         assembly ("memory-safe") {
113             if iszero(d) {
114                 mstore(0x00, 0x65244e4e) // `DivFailed()`.
115                 revert(0x1c, 0x04)
116             }
117             z := sdiv(x, d)
118             if sgt(z, 0) {
119                 z := add(z, iszero(iszero(smod(x, d))))
120             }
```

```
121             }
122         }
```

Other functions have similar issues, for example:

- `PMath.mulCeil(int256 x, int256 y)` : returns `0` instead of `1` when $0 < x \times y < 10^{18}$
- `PMath.rawDivFloor(int256 x, int256 d)` : returns `0` instead of `-1` when $-1 < \frac{x}{d} < 0$
- `PMath.mulFloor(int256 x, int256 y)` : returns `0` instead of `-1` when $-10^{18} < x \times y < 0$

Status

✓ Fixed

[WP-L15] The residual assets from ”remainder of division when converting scaled amount to unscaled during withdrawal `scaledAmount % scalingFactor` ” accumulate and become locked in the MarketHub contract.

Low

Issue Description

For high-value assets like BTC or tokens with very small decimals like Gemini dollar (GUSD whose decimals is 2), these accumulated remainders could lock a significant amount of assets in the contract over time.

Consider modifying `cashFeeData[tokenIds[i]].treasuryCash` to only subtract the evenly divisible part `_toUnscaled(scaledAmount, data.scalingFactor) * data.scalingFactor` , keeping the remainder in `cashFeeData[tokenIds[i]].treasuryCash` .

Line 103 could be changed to: `cashFeeData[tokenIds[i]].treasuryCash = scaledAmount - unscaledAmount * data.scalingFactor;` for accurate tracking.

```
100     function withdrawTreasury(TokenId[] memory tokenIds) external onlyAuthorized {
101         for (uint256 i = 0; i < tokenIds.length; i++) {
102             uint256 scaledAmount = cashFeeData[tokenIds[i]].treasuryCash;
103             cashFeeData[tokenIds[i]].treasuryCash = 0;
104 
105             TokenData memory data = _tokenDataChecked(tokenIds[i]);
106             IERC20(data.token).safeTransfer(TREASURY, _toUnscaled(scaledAmount,
    data.scalingFactor));
107 
108             emit CollectFee(tokenIds[i], scaledAmount);
109         }
110     }
```

```
146     function _toUnscaled(uint256 scaledAmount, uint96 scalingFactor) internal pure
    returns (uint256) {
147         return scaledAmount / scalingFactor;
148     }
```

Status

ⓘ Acknowledged

[WP-M16] `cancelVaultWithdrawal()` can be reentered when the `token` has hooks

Medium

Issue Description

The `finalizeVaultWithdrawal()` function can be used to reenter `cancelVaultWithdrawal()`, allowing an attacker to withdraw `_withdrawal[root][tokenId].unscaled` while also increasing cash for `root` (resulting in `root` receiving an extra amount of `_tokenDataChecked(tokenId).token` equal to `_withdrawal[root][tokenId].unscaled`).

This is possible because on line L147, `IERC20(data.token).safeTransfer(root, user.unscaled)` is executed before updating the `_withdrawal[root][tokenId]` state, and there's no reentrancy protection.

```
128     function cancelVaultWithdrawal(address root, TokenId tokenId) external
    onlyRouter {
129         TokenData memory data = _tokenDataChecked(tokenId);
130         MarketAcc main = root.toMainCross(tokenId);
131         Withdrawal memory user = _withdrawal[root][tokenId];
132 
133         _topUpWithdrawCash(main, _toScaled(user.unscaled, data.scalingFactor),
    true);
134         emit VaultWithdrawalCanceled(root, tokenId, user.unscaled);
135 
136         user.unscaled = 0;
137         _withdrawal[root][tokenId] = user;
138     }
139 
140     /// @dev allow this to be called by anyone
141     function finalizeVaultWithdrawal(address root, TokenId tokenId) external {
142         TokenData memory data = _tokenDataChecked(tokenId);
143         Withdrawal memory user = _withdrawal[root][tokenId];
144 
145         require(uint256(user.start) + uint256(_getPersonalCooldown(root)) <=
    block.timestamp, Err.MHWithdrawNotReady());
146 
147         IERC20(data.token).safeTransfer(root, user.unscaled);
148         emit VaultWithdrawalFinalized(root, tokenId, user.unscaled);
```

```
149
150         user.unscaled = 0;
151         _withdrawal[root][tokenId] = user;
152     }
```

```
57      function _topUpWithdrawCash(MarketAcc payer, uint256 scaled, bool isDeposit)
    internal {
58          if (isDeposit) {
59              acc[payer].cash += scaled.Int();
60          } else {
61              acc[payer].cash -= scaled.Int();
62              _checkIMStrict(payer);
63          }
64      }
```

Status

✓ Fixed

Appendix

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by WatchPug; however, WatchPug does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication.

Disclaimer

This report is based on the scope of materials and documentation provided for a limited review at the time provided. Results may not be complete nor inclusive of all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Smart Contract technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. A report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.