
INDEPENDENT SECURITY REVIEW

MNX

Security Review

PREPARED FOR

MNX

LEAD RESEARCHER

Sujith Somraaj (sujith@centennial.systems)

ISSUED

June 23, 2026 - July 23, 2026

Contents

1	About Centennial	2
2	Disclaimer	2
3	Scope	2
4	Risk Classification	2
4.1	Impact	3
4.2	Likelihood	3
4.3	Action Required	3
5	Executive Summary	4
6	Findings	5
6.1	High Risk	5
6.2	Medium Risk	10
6.3	Low Risk	30
6.4	Informational	49
7	Notes from the Researcher	60

1 About Centennial

Centennial Systems is a boutique product, security, and engineering practice for founders, operators, and enterprises building financial infrastructure. It partners with teams where subtle failures can carry material operational, security, or commercial consequences, combining strategic direction with hands-on execution.

Centennial works across three closely connected disciplines. Its product practice supports roadmap, prioritization, launch, and go-to-market decisions. Its security work includes adversarial architecture reviews, security-driven system design, organizational security policies, fractional CISO leadership, and operational security. Its engineers embed with teams across crypto, fintech, and AI systems to take scoped work through production and handover.

Small by design, Centennial works directly with founders and operators, scopes each engagement around the problem at hand, and stays involved through delivery. Learn more at centennial.systems.

2 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of that given smart contract(s) or blockchain software. i.e., the evaluation result does not guarantee against a hack (or) the non existence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, I always recommend proceeding with several audits and a public bug bounty program to ensure the security of smart contract(s). Lastly, the security audit is not an investment advice.

This review is done independently by one reviewer.

3 Scope

REVIEWED COMPONENT

contracts/**/*sol

4 Risk Classification

Severity is assessed by combining the potential impact of a finding with the likelihood that it can be triggered under realistic conditions.

LIKELIHOOD / IMPACT	HIGH	MEDIUM	LOW
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

4.1 Impact

LEVEL	DEFINITION
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses below 10%, or losses to only a subset of users, that remain unacceptable.
Low	Bearable or readily repairable losses, including grieving and gas inefficiencies.

4.2 Likelihood

LEVEL	DEFINITION
High	Almost certain to occur, easy to perform, or difficult but highly incentivized.
Medium	Conditionally possible or incentivized, but still relatively likely.
Low	Requires multiple conditions to align, with little or no incentive.

4.3 Action Required

SEVERITY	REQUIRED RESPONSE
Critical	Must be fixed as soon as possible if already deployed.
High	Must be fixed before deployment, if not already deployed.
Medium	Should be fixed.
Low	Could be fixed.

5 Executive Summary

REVIEW NOTE

Over the course of 25 days, MNX engaged the researcher to review the contracts defined in the scope of this document. This was a differential review, limited to changes introduced in the audit commit. A total of 82 findings were recorded.

ENGAGEMENT

Project	MNX
Repository	mnx-markets/exchange-contracts
Commit	8f4cebe4
Timeline	June 23, 2026 - July 23, 2026
Method	Manual Review
Documentation	Low-Medium
Test coverage	Medium

FINDINGS

82	
TOTAL RECORDED	
Critical	0
High	2
Medium	24
Low	35
Gas	0
Informational	21

6 Findings

6.1 High Risk

6.1.1 Delisted child positions use live rather than frozen settlement values

Context: [LiquidityVault.sol#L676-L683](#), [Perpetual.sol#L1783-L1797](#), [Perpetual.sol#L1229-L1247](#), [Perpetual.sol#L2499-L2522](#)

Description: `LiquidityVault.totalNAV()` obtains each child position's value through `accountFundingAdjustedEquity()`. That view always reads the current oracle price and settles funding against the stored `globalIndex`.

Once a Perpetual is delisted, however, the amount that the child can actually receive is fixed using `delistingPrice` and `delistIndex`. Closing the position later cannot realize a value calculated from a different oracle price or funding index. The price mismatch affects both delisting paths whenever the oracle moves after delisting. An index mismatch additionally arises in the explicit-price path because `delistPerpetualAtSettlementPrice()` records a settlement index without updating `globalIndex`.

For example, assume the main vault holds 100 tokens and a child has a one-unit long opened at 100 with 20 tokens of margin. The market is delisted at 100, so the child's frozen settlement value is 20 and the vault's true settlement NAV is 120. If the live oracle later reports 150, `accountFundingAdjustedEquity()` reports the child as worth 70, causing `totalNAV()` to report 170.

An LP redeeming 50 of the vault's 100 shares is then paid 85 tokens, even though half of the frozen settlement NAV is only 60 tokens. After the position settles for 20 tokens, only 35 tokens remain for the other 50 shares rather than 60. The exiting LP has taken 25 tokens from the remaining LPs based on value the delisted position can never realize.

Recommendation: When `delisted == true`, calculate account equity using the stored `delistingPrice` and `delistIndex`, matching the exact context used by `closePosition()`. This can be implemented by making `accountFundingAdjustedEquity()` branch to frozen settlement accounting or by exposing a separate settlement-equity view that `LiquidityVault` uses. As an additional safeguard, pause vault deposits and withdrawals involving a delisted market until its child positions have been settled and reconciled.

MXN: Fixed in [6c99da](#).

Once a market is delisted, account equity now uses the frozen delisting price and delisting funding index—the same settlement context used by `closePosition()`—rather than the live oracle and global index. `LiquidityVault` therefore values a child position at the amount it can actually settle.

The close-path refactor was checked with a 24-case execution-level differential test that runs the old and new close paths against each other, and the suite covers both delisting paths, post-delist oracle movement, funding-index differences, profitable positions, losing positions, and bankruptcy.

Researcher: Verified fix.

6.1.2

LP redemptions priced against NAV including unrealized PnL let exiting LPs extract paper gains from remaining LPs

Context: [LiquidityVault.sol#L668](#)

Description: `deposit` and `executeWithdrawal` price shares against `totalNAV()`, which includes each child's mark-to-market unrealized PnL (`accountFundingAdjustedEquity` → `_positionEquity` = `margin` + `_positionPnl` at the live `_readOraclePrice()`), yet withdrawals pay out real collateral from the main vault.

An LP can therefore redeem at a NAV inflated by unrealized (unsettled, reversible) profit and convert paper gains into actual tokens, depleting the pool's real backing.

Concretely, with A and B each depositing `1e18` and child positions running `+2e18` unrealized PnL, A withdraws his 50% of the `4e18` NAV as `2e18` real tokens, draining the pool's real cash; when the PnL reverses, B who deposited `1e18` is left holding shares worth `~0`, a clean `1e18` transfer from the passive LP to the exiting one.

The withdrawal timelock does not prevent this because `executeWithdrawal` prices at execution-time NAV, giving exiting LPs a free option to fire when the mark is high; the symmetric case lets new depositors enter at a depressed NAV and dilute incumbents.

The only natural bound is the insufficient main liquidity check, which caps extraction to idle main-vault cash but does not correct the direction of harm.

PoC:

The following foundry test case demonstrates the issue.

- Active LP deposits `1e18` and requests withdrawal while NAV is `2e18`.
- Passive LP deposits `1e18`.
- Mock child market reports `+2e18` mark-to-market equity, pushing `totalNAV()` to `4e18`.
- Active LP executes the already-requested withdrawal and receives `2e18`, draining the main vault's idle balance.
- The mark is removed, and the passive LP still has shares but `totalNAV()` reverts as non-positive.

```
pragma solidity ^0.8.26;

import "forge-std/Test.sol";

import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

import {LiquidityVault} from "../contracts/LiquidityVault.sol";
import {MarginBank} from "../contracts/MarginBank.sol";
import {AccountManager} from "../contracts/AccountManager.sol";

import {SignatureVerifier} from "../contracts/SignatureVerifier.sol";

import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";

contract CollateralToken is ERC20 {
    constructor() ERC20("CollateralToken", "CTT") {
        _mint(msg.sender, 1_000_000e18);
    }
}

contract AuditTest is Test {
```

```

address trustedForwarder = makeAddr("TRUSTED_FORWARDER");
address owner = makeAddr("OWNER");
address guardian = makeAddr("GUARDIAN");
address user = makeAddr("USER");

address market = makeAddr("MARKET");
address childVault;

LiquidityVault liquidityVault;
AccountManager accountManager;
MarginBank marginBank;

CollateralToken token;
SignatureVerifier signatureVerifier;

LiquidityVaultHarness harness;

function setUp() external {
    vm.startPrank(user);
    token = new CollateralToken();
    vm.stopPrank();

    signatureVerifier = new SignatureVerifier();

    vm.startPrank(owner);
    AccountManager accountManagerImpl = new AccountManager();
    ERC1967Proxy proxy = new ERC1967Proxy(address(accountManagerImpl),
    ↪ abi.encodeWithSelector(AccountManager.initialize.selector, trustedForwarder));
    accountManager = AccountManager(address(proxy));

    MarginBank marginBankImpl = new MarginBank();
    proxy = new ERC1967Proxy(address(marginBankImpl), abi.encodeWithSelector(MarginBank.initialize.selector,
    ↪ address(token), address(guardian), address(trustedForwarder), address(signatureVerifier)));
    marginBank = MarginBank(address(proxy));

    LiquidityVault implementation = new LiquidityVault();
    proxy = new ERC1967Proxy(address(implementation), abi.encodeWithSelector(LiquidityVault.initialize.selector,
    ↪ address(marginBank), address(accountManager), address(token), owner, guardian, 1_000, 1));
    liquidityVault = LiquidityVault(address(proxy));

    marginBank.setSystemAccountController(address(liquidityVault), true);
    accountManager.setSystemSubAccountController(address(liquidityVault), true);
    childVault= liquidityVault.bootstrapVaultAccounts(
        bytes32("mainLabel"),
        bytes32("mainLabel"),
        user
    );
    liquidityVault.setVaultDepositWhitelistEnabled(false);
    liquidityVault.markChildObserved(childVault);
    liquidityVault.setChildActiveForRouting(childVault, true);
    liquidityVault.registerMarket(market, true);
    vm.stopPrank();

    // Bare harness (never initialized) used purely to fuzz the internal
    // _recordVaultWithdrawal rounding math in isolation.
    harness = new LiquidityVaultHarness();
}

```

```

function test_poc_withdrawal_executes_against_unrealized_mark_and_drains_main_liquidity()
    external
{
    address activeLp = makeAddr("ACTIVE_LP");
    address passiveLp = makeAddr("PASSIVE_LP");
    MockVaultMarket markMarket = new MockVaultMarket();

    vm.startPrank(owner);
    liquidityVault.registerMarket(address(markMarket), true);
    liquidityVault.setChildActiveForMarket(childVault, address(markMarket), true);
    vm.stopPrank();

    vm.startPrank(user);
    token.transfer(activeLp, 1e18);
    token.transfer(passiveLp, 1e18);
    vm.stopPrank();

    vm.startPrank(activeLp);
    token.approve(address(liquidityVault), 1e18);
    uint256 activeShares = liquidityVault.deposit(1e18);
    bytes32 withdrawId = liquidityVault.requestWithdrawal(activeShares);
    vm.stopPrank();

    vm.startPrank(passiveLp);
    token.approve(address(liquidityVault), 1e18);
    uint256 passiveShares = liquidityVault.deposit(1e18);
    vm.stopPrank();

    assertEq(activeShares, 1e18);
    assertEq(passiveShares, 1e18);
    assertEq(liquidityVault.totalNAV(), 2e18);
    assertEq(marginBank.getAccountBankBalance(liquidityVault.mainVaultAddress()), 2e18);

    // The child has no realized/free tokens in the main vault, but NAV now
    // includes the child market's positive mark-to-market equity.
    markMarket.setEquity(childVault, 2e18);
    assertEq(liquidityVault.totalNAV(), 4e18);

    vm.warp(block.timestamp + 2);

    uint256 activeBalanceBefore = token.balanceOf(activeLp);
    vm.prank(activeLp);
    uint128 paid = liquidityVault.executeWithdrawal(withdrawId);

    assertEq(paid, 2e18);
    assertEq(token.balanceOf(activeLp) - activeBalanceBefore, 2e18);
    assertEq(marginBank.getAccountBankBalance(liquidityVault.mainVaultAddress()), 0);
    assertEq(liquidityVault.sharesOf(passiveLp), passiveShares);

    // When the mark disappears, the passive LP is left with shares but no
    // main-vault liquidity backing them.
    markMarket.setEquity(childVault, 0);
    vm.expectRevert(bytes("LiquidityVault: NAV non-positive"));
    liquidityVault.totalNAV();
}
}

/// Test-only subclass exposing the internal reduction math and the two

```

```

/// storage trackers so the fuzzer can explore principal != shares directly.
contract LiquidityVaultHarness is LiquidityVault {
    function h_setNetDeposits(address depositor, uint128 amount) external {
        netVaultDeposits[depositor] = amount;
    }

    function h_setTotalNetDeposits(uint128 amount) external {
        totalNetVaultDeposits = amount;
    }

    function h_recordWithdrawal(
        address depositor,
        uint256 sharesBurned,
        uint256 sharesBefore
    ) external {
        _recordVaultWithdrawal(depositor, sharesBurned, sharesBefore);
    }
}

contract MockVaultMarket {
    mapping(address => int256) public equityOf;

    function setEquity(address account, int256 equity) external {
        equityOf[account] = equity;
    }

    function accountFundingAdjustedEquity(address account) external view returns (int256) {
        return equityOf[account];
    }
}

```

Recommendation: Price redemptions against realized value - exclude or haircut unrealized PnL from the redemption NAV (or require settlement/realization first), and price withdrawals at request-time NAV (or min(request, execute)) to remove the execution-time timing option; optionally add a redemption fee/entry-exit spread so round-trip timing is not free.

MNX: Acknowledged. Deferred until after the LiquidityVault share-price redesign. That redesign will ship before the vault is opened to public depositors.

Researcher: Acknowledged.

6.2 Medium Risk

6.2.1 A minimum-size trade can pin maximum funding across otherwise empty windows

Context: [Perpetual.sol#L650-L657](#), [Perpetual.sol#L790-L796](#), [Perpetual.sol#L1604-L1613](#), [FundingOracle.sol#L181-L218](#), [FundingOracle.sol#L263-L316](#), [Evaluator.sol#L240-L255](#)

Description: Every non-empty regular settlement batch contributes exactly one funding observation at its uniform clearing price. `Perpetual.trade()` retains only that price and passes it to `FundingOracle.recordTrade()`; neither the batch's total quantity nor its notional is included. Consequently, a minimum-quantity trade has the same effect on the funding ratio as a batch containing the market's entire legitimate volume.

`recordTrade()` replaces `currentFundingWindow.ratio` with the latest trade premium. When `setFundingRate()` closes the window, it resets `sum`, but copies that ratio into the new window:

```
currentFundingWindow = FundingWindow({
  tFirst: currentTimestamp,
  tLast: currentTimestamp,
  sum: 0,
  ratio: currentFundingWindow.ratio
});
```

If no later trade occurs, each elapsed second in every subsequent window is therefore treated as another observation at the last trade's premium. The sample has no volume-dependent weight, freshness limit, or expiry.

For example, assume the oracle price is 100, `maxFunding` is the documented 0.1% per hour, and the configured price bounds permit a fill at 110. Two attacker-controlled addresses can sign opposing minimum-quantity orders filled at 110. The pair is collectively market-neutral apart from fees, but the batch records a +10% premium. Since $10\% / 24$ exceeds the configured hourly cap, the next scheduled update installs the maximum positive rate:

$$0.1\% / 3,600 = 277777777777 \text{ per second}$$

After `setFundingRate()` resets the window, the retained +10% ratio fills the entire next hour despite there being no trades. The following scheduled update installs exactly the same maximum rate. This repeats across every empty window until another regular batch changes the ratio or funding is disabled. A focused Foundry test reproduced the same 277777777777 rate in two consecutive full-hour windows while `windowSet` advanced normally.

An attacker can hold a large short position separately and use the small opposing-order pair only to establish the sample. Longs across the market then continue paying the attacker's short at the maximum rate, so the benefit scales with aggregate open interest and duration while the manipulation cost is limited to minimum-size trade fees and collateral. The negative-premium case is symmetric for a large long.

This does not require the missed keeper update or one-second observation window described in *issue A missed funding update can create a one-second observation window*: rate updates may occur on schedule and every observation window may span a full hour.

Submitting `Perpetual.trade()` requires a settlement operator. Exploitation by an ordinary user therefore depends on the off-chain matcher accepting two commonly controlled crossing orders at the selected price. A faulty or compromised settlement operator can include them directly. If the operator is assumed to reliably reject wash trades and exercise price-quality discretion

beyond the on-chain rules, this is instead an architectural trust dependency. On-chain, however, the orders may satisfy all signature, minimum-quantity, limit-price, market-take-bound, and account-health checks, and different addresses are not treated as a self-trade.

Recommendation: Incorporate executed quantity or notional into funding observations and compute a volume-weighted or otherwise manipulation-resistant mark price. Require a meaningful minimum aggregate notional or sample count before a window can install a nonzero trade-derived rate.

Do not indefinitely carry the last ratio through inactive windows. Expire observations after a bounded age and reset the ratio to zero, or fall back to a robust reference mark, when a window contains insufficient fresh trading activity. Self-trade or common-control filtering should be added as defense in depth, but should not be the sole protection because colluding accounts and a compromised settlement operator can bypass identity-based filtering.

MXN: The vulnerable contract mechanism was removed in [d2c92a29](#), with final contract hardening in [c48d8599](#) and contract regression coverage in [02329a38](#).

Researcher: Verified fix.

6.2.2

Vault share pricing omits elapsed funding until the global index is checkpointed

Context: [LiquidityVault.sol#L419-L445](#), [LiquidityVault.sol#L512-L562](#)

Description: `LiquidityVault.deposit()` and `executeWithdrawal()` mint and redeem shares using the value returned by `totalNAV()`. For each active child position, `totalNAV()` calls `Perpetual.accountFundingAdjustedEquity()`.

That view settles the child only against the already stored `globalIndex.value`. It does not project the index from `globalIndex.timestamp` to the current block time. Elapsed funding enters the child's reported equity only after somebody calls the `permissionlessUpdateGlobalIndex()` function.

An LP can therefore change its ownership of the vault against NAV that omits a funding liability or credit which has already economically accrued.

For a pending funding debit, a contract-owned LP can atomically execute a matured withdrawal and then call `updateGlobalIndex()`. The withdrawal is paid using the stale, higher NAV. The newly recognized debit is then borne entirely by the LPs whose shares remain. No transaction can be inserted between the two calls to correct the ordering.

The symmetric case applies to deposits when the child has a pending funding credit. A depositor mints shares against the stale, lower NAV, checkpoints the index, and shares in funding that accrued before the depositor joined, diluting incumbent LPs.

A focused Foundry test demonstrates the withdrawal case using the real vault, `MarginBank`, and `Perpetual` funding-accounting paths:

- Alice and Bob each deposit 10,000 tokens and receive 10,000 shares.
- The vault child holds a 100,000-notional long with 10,000 margin.
- Funding is set to the documented 0.1%-per-hour maximum.
- After one hour, the child has an uncheckpointed funding debit of approximately 100 tokens.
- Executing Alice's withdrawal before the checkpoint pays her the stale amount of 10,000 tokens.
- The subsequent checkpoint reduces the remaining vault NAV to approximately 9,900, assigning the entire debit to Bob.
- If the checkpoint occurs first, Alice receives approximately 9,950 and Bob retains approximately 9,950, correctly splitting the debit between them.

The atomic ordering used by the test is:

```
function exitThenCheckpoint(
    LiquidityVault vault,
    bytes32 withdrawalId,
    Perpetual market
) external returns (uint128 paid) {
    paid = vault.executeWithdrawal(withdrawalId);
    market.updateGlobalIndex();
}
```

The relevant assertions are:

```
assertEq(stalePaid, 10_000e18);
assertEq(vault.totalNAV(), 9_900_000000000280000000);

// Control execution: checkpoint before redeeming the same shares.
assertEq(freshPaid, 9_950_000000000140000000);
assertEq(vault.totalNAV(), 9_950_000000000140000000);
```

Both focused tests pass:

```
[PASS] test_atomicExitBeforeCheckpointExtractsElapsedFundingFromRemainingLp()
[PASS] test_checkpointFirstWouldLimitTheSameWithdrawalToFifty()
2 passed; 0 failed
```

The transferable amount grows with the checkpoint delay, aggregate child exposure, and funding rate. It can be repeated whenever vault share changes are accepted while one or more child-market indices trail wall-clock time.

The child remains the position holder throughout and the eventual funding debit is assigned to the correct trading account. The incorrect transfer occurs because LP share ownership changes before that account's already elapsed funding is incorporated into vault NAV.

Recommendation: Ensure every active child market's funding index is current before calculating NAV for a share mint or burn. The checkpoint must happen inside the same deposit or withdrawal transaction before `totalNAV()` is read; relying on a separate keeper transaction leaves the ordering exploitable.

Alternatively, expose a read path that safely projects each market's funding index through the current timestamp and use that projected equity in vault NAV. Enforce a strict maximum index age if projection is not possible. When multiple children use the same market, update or project that market once and consistently apply the resulting index to every child.

MNX: This issue is deferred until the LiquidityVault share-pricing redesign is complete. The redesign will ship before LiquidityVault deposits are opened to the public.

Researcher: Acknowledged.

6.2.3 Binary-short backstop trigger uses spot notional and can inherit healthy accounts

Context: [PositionBalanceMath.sol#L135-L170](#), [Perpetual.sol#L1526-L1539](#), [Perpetual.sol#L1939-L1975](#), [Perpetual.sol#L2030-L2128](#), [Perpetual.sol#L2638-L2705](#)

Description: Perpetual deliberately measures the risk of a short binary position using its remaining payout exposure. `getMarginRatioWithBinaryCheck()` divides equity by $qPos - qPos * price$. This correctly reflects that a short's maximum remaining loss approaches zero as the outcome price approaches one.

The backstop trigger does not use the same risk basis. `_attemptBackstopInheritance()` calculates maintenance margin as:

```
position.qPos * context.price * maintenanceMarginRequired
```

It then permits inheritance whenever positive equity is below a configured fraction of that amount. For a binary short, this uses the position's current spot value rather than its remaining payout risk. The two values move in opposite directions near the upper payout bound: canonical risk approaches zero while the backstop notional approaches the full quantity.

For example, consider a one-unit binary short opened and currently priced at 0.9, with `oiOpen = 0.9`, margin of 0.01, IMR of 10%, MMR of 5%, and a backstop trigger fraction of 25%. Its equity is 0.01, remaining payout risk is 0.1, and its canonical margin ratio is therefore 10%. The account meets IMR and is healthy under the same binary-aware check used by the trading path.

The backstop path instead calculates maintenance as $1 * 0.9 * 5\% = 0.045$ and the trigger as $0.045 * 25\% = 0.01125$. Because $0.01 < 0.01125$, the settlement operator can successfully inherit the healthy position. `_transferPositionToChildVault()` then clears the user's position and margin while assigning both to the vault child.

A focused Foundry test using the real inheritance path confirms the mismatch:

```
assertEq(perpetual.getMarginRatio(ALICE), 0.1e18);
assertTrue(perpetual.attemptBackstopInheritance(ALICE));

Types.PositionBalance memory userAfter = perpetual.getAccountBalance(ALICE);
Types.PositionBalance memory childAfter = perpetual.getAccountBalance(CHILD);

assertEq(userAfter.qPos, 0);
assertEq(userAfter.margin, 0);
assertEq(childAfter.qPos, 1e18);
assertEq(childAfter.margin, 0.01e18);
```

The issue does not require the owner to configure the maximum trigger fraction. For any nonzero fraction, a sufficiently high binary outcome price can make the spot-notional trigger exceed the maintenance requirement derived from the remaining payout risk.

An authorized settlement operator is required, but the role should not be able to seize an account that the protocol's canonical risk checks classify as healthy.

Recommendation: Calculate backstop eligibility from the same binary-aware risk notional used by `getMarginRatioWithBinaryCheck()`. Prefer comparing the canonical margin ratio directly against `maintenanceMarginRequired * backstopTriggerMarginFractionE18`, so ordinary and binary markets share one health definition. Add regression tests proving that accounts at or above IMR or MMR cannot be inherited across binary prices near both payout bounds.

MXN: Fixed across the shared risk-notional primitive in [17a7084](#) and its market integration in [6cff9da](#).

Researcher: Verified fix.

6.2.4 Backstop inheritance confiscates a liquidatable account's positive residual equity

Context: [Perpetual.sol#L336-L340](#), [Perpetual.sol#L1511-L1539](#), [Perpetual.sol#L1939-L2028](#), [Perpetual.sol#L2030-L2190](#), [IsolatedTrader.sol#L643-L692](#)

Description: Backstop inheritance is described as the fallback for a distressed position that cannot be cleanly closed through normal liquidation. The on-chain trigger does not establish that condition. It accepts any position satisfying:

```
0 < equity < maintenanceMargin * backstopTriggerMarginFractionE18
```

Because the owner may set the fraction as high as $1e18$, every positive-equity account below maintenance margin can be selected. Such an account is also eligible for ordinary liquidation, and the settlement operator can call `attemptBackstopInheritance(account)` directly without first attempting or failing that ordinary path.

Inheritance then transfers more than the position's risk. `_mergeInheritedPosition()` adds the user's full equity to the selected child. For an empty child, it reopens the same quantity at the current price and sets the child's margin equal to the user's positive equity. `_transferPositionToChildVault()` subsequently stores an empty position for the user. No MarginBank transfer compensates the user because `childFreeCollateralDelta` is zero in this case.

The economic result differs from an ordinary reduction. Under `IsolatedTrader._applyIM()`, a one-unit long opened at 100 with four tokens of margin and reduced at 100 with zero fee produces `fundsFlow = -4`, returning four tokens to the user's bank balance. With a 5% maintenance requirement and a 100% backstop trigger fraction, the same account has positive equity of four and maintenance of five, so the operator may instead invoke inheritance and route all four tokens to the LP-owned child.

A focused Foundry test using the real Perpetual inheritance path confirms the transfer:

```
assertEq(perpetual.accountFundingAdjustedEquity(ALICE), 4e18);
assertTrue(perpetual.attemptBackstopInheritance(ALICE));

Types.PositionBalance memory userAfter = perpetual.getAccountBalance(ALICE);
Types.PositionBalance memory childAfter = perpetual.getAccountBalance(CHILD);

assertEq(userAfter.qPos, 0);
assertEq(userAfter.margin, 0);
assertEq(childAfter.qPos, 1e18);
assertEq(childAfter.margin, 4e18);
assertEq(perpetual.accountFundingAdjustedEquity(CHILD), 4e18);
```

An authorized settlement operator is required, but that role otherwise does not have a general right to transfer an account's collateral to the LP vault. The route choice therefore lets a faulty or compromised operator decide whether a liquidatable user's residual is returned or confiscated. The maximum transfer is owner-configurable up to the account's full positive equity immediately below maintenance margin.

Recommendation: Define an explicit acquisition price and liquidation fee for inherited positions. Return the user's residual equity after a bounded fee, with the vault posting any collateral required to carry the inherited position. Alternatively, restrict uncompensated inheritance to non-positive-equity positions and require the ordinary liquidation path to be exhausted for positive-equity accounts. The settlement operator should not be able to choose between economically different outcomes for the same eligible account without an objective on-chain condition.

MNX: Reviewed and documented as intended protocol policy in [abcc07e](#).

Researcher: Acknowledged.

6.2.5

Trading halt does not stop irreversible global funding-index updates

Context: [Perpetual.sol#L443-L456](#), [Perpetual.sol#L1604-L1659](#), [Perpetual.sol#L2359-L2397](#), [Perpetual.sol#L2447-L2487](#)

Description: whenTradingAllowed is applied to trades, margin/leverage operations, direct backstop attempts, and permissionless per-account funding settlement. Its comment specifically explains that a halt may be used while remediating a bad funding-rate submission so users and keepers cannot bake the halted index into account state.

updateGlobalIndex() and setFundingRate() omit the modifier. Both are permissionless, and both call _loadContext(true), which reads the still-live funding rate and permanently advances globalIndex.value through _applyFundingToGlobalIndex(). setOffChainFundingRate() has the same unguarded checkpoint before installing its replacement rate, although its caller is privileged.

An arbitrary caller can therefore wait until the guardian halts trading in response to a bad rate and call updateGlobalIndex(). Per-account settlement is correctly blocked by P7 during the halt, but the bad rate has already been irreversibly incorporated into the global index. When trading resumes, accounts realize that poisoned index normally. The contract exposes no rewind or correction for globalIndex.value, so blocking only the final per-account write does not preserve the remediation window promised by the guard.

A focused Foundry test configured a one-unit long with 20 tokens of margin, a price of 100, a funding rate of 1e14 per second, and globalIndex.timestamp = 1000. After the guardian returned false, it advanced time to 1100:

```
guardian.setTradingAllowed(false);
vm.warp(1_100);

perpetual.updateGlobalIndex();
perpetual.setFundingRate();

(uint128 timestamp, int256 value) = perpetual.globalIndex();
assertEq(timestamp, 1_100);
assertEq(value, 1e18);

vm.expectRevert(bytes("P7"));
perpetual.settleFundingForAccounts(accounts);

guardian.setTradingAllowed(true);
perpetual.settleFundingForAccounts(accounts);
assertEq(perpetual.getAccountBalance(ALICE).margin, 19e18);
```

Both unguarded calls succeeded during the halt. Once the guardian resumed trading, the account lost one token to the index advanced while remediation was supposed to be in progress.

This is **novel relative to Funding-input transitions retroactively reprice or erase already elapsed funding**. Issue *Funding-input transitions retroactively reprice or erase already elapsed funding* concerns changing an oracle, funder, or trading/funding lifecycle input without first checkpointing the old regime. Here no input transition is required: a public caller defeats an already-active emergency halt by advancing the existing bad input into irreversible global state.

Recommendation: Apply whenTradingAllowed to the permissionless updateGlobalIndex() and setFundingRate() paths. Do not merely apply the same modifier to every corrective operator path: the protocol still needs a privileged, explicit remediation function that can install a corrected rate and cutoff without first applying the bad rate over the halted interval. Define whether

funding freezes at the halt timestamp or continues under a separately approved rate, checkpoint the last known-good interval atomically, and make resumption commit to that corrected index.

MNX: The first layer is in [6cff9da](#): during a Guardian trading halt, every path that realizes the global funding index into any account's state fails with the halt error — trading, settlement-operator operations, margin operations, and the permissionless funding-settlement call — so a suspect rate cannot reach account balances while remediation is in progress.

The second layer is in [e8999b4](#) and closes the finding at the index itself: permissionless index advancement (`updateGlobalIndex` and the funding-rate poke) now also fails with the halt error during a trading halt, with one narrow exemption for the Guardian contract as caller — its own funding-status transitions checkpoint the elapsed interval in both directions.

The gate reads the market's trading status from local storage rather than through the guardian, so a replaced guardian contract cannot widen the exemption into a bypass. This layer matters because the owner's reviewed-cutoff path (described in issue with title: Funding-input transitions retroactively reprice or erase already elapsed funding) bounds its cutoff below by the stored index timestamp: a mid-halt advance would ratchet that floor forward and put the suspect interval permanently beyond re-pricing. With the gate, the index freezes at the halt and the reviewed cutoff keeps its full reach. Funding is not lost during a halt unrelated to funding — it accrues lazily, and the first post-resume advance prices the elapsed span at the stored rate.

One deliberate exception is disclosed: the operator-privileged off-chain funding-rate path stays available during a halt because it is part of the rate-correction toolkit; it is a single authorized key, not a permissionless surface. Tests cover the race in both directions (a halted advance is rejected and the cutoff is accepted at the pre-halt timestamp; an unhalted advance does move the floor), both guardian flip directions, the dead-feed halt-entry path, and post-resume pricing of the full halted span.

Researcher: Verified fix.

6.2.6 Rounded-down BPS calculation allows updates above the advertised oracle cap

Context: [BackendPriceOracle.sol#L102-L117](#), [Perpetual.sol#L1882-L1924](#)

Description: `BackendPriceOracle.setPrice()` converts an update's percentage movement to whole basis points before comparing it with `maxUpdateBps`:

```
bps = diff * 10_000 / previousPrice
```

Solidity division rounds down. Consequently, a configured cap of C basis points accepts movements approaching $C + 1$ basis points. With a previous answer of `100e18` and a 100-bps cap, an answer of `101.009e18` moves by 100.9 bps, but the contract computes `floor(100.9) = 100` and accepts it.

This is not only a display discrepancy. Perpetual treats `maxUpdateBps` as an exact input to its ADL replacement-capacity proof. For a 10% initial margin, a 10.01% long market-take bound, and an advertised 1% oracle move, P57 evaluates:

```
1.01 * 1.1001 * 0.9 = 0.9999909 <= 1
```

The 1.009% move accepted by the oracle instead gives:

```
1.01009 * 1.1001 * 0.9 = 1.0000800081 > 1
```

Thus a configuration accepted as solvent at the advertised cap can receive a valid oracle update that violates the proof. The discrepancy is less than one basis point per update, but its collateral impact scales with open notional and configurations are allowed to sit arbitrarily close to the P57/P58 boundary.

Recommendation: Compare the unrounded products directly, for example:

```
require(uint256(diff) * 10_000 <= uint256(previousPrice) * maxUpdateBps, "BP0: change too large");
```

Alternatively, round the computed basis-point movement up before comparison. Add boundary tests for movements exactly at the cap, just above it, and for configurations whose P57/P58 result has less than one basis point of headroom.

MX: Fixed in [9d44dec](#). `BackendPriceOracle` now compares `diff * 10_000 <= previousPrice * maxUpdateBps` directly instead of rounding the movement down to whole basis points. The exact configured boundary remains valid and one raw price unit above it is rejected, with regression coverage for the fractional-basis-point case.

Researcher: Verified fix.

6.2.7

The ADL capacity proof does not cover the actual oracle-to-risk interval

Context: [Perpetual.sol#L509-L515](#), [Perpetual.sol#L1887-L1924](#), [BackendPriceOracle.sol#L102-L142](#)

Description: Perpetual's P57/P58 checks prove that the configured initial margin and market-take bounds can tolerate one oracle movement of `maxUpdateBps`. The proof is useful only if that advertised movement also bounds the price change between the last risk-processing opportunity and the next liquidation or ADL.

The deployed contracts do not enforce that relationship. `BackendPriceOracle` limits each individual round, while Perpetual reads only the latest answer and neither records which round was processed nor acknowledges a round before the next one can be published. Two individually valid 1% increases can therefore compound into a 2.01% change before any position is liquidated or deleveraged. `lastUpdateBlock` permits another round in the next block, and `minUpdateIntervalSeconds` is optional; neither proves that liquidation or ADL ran between rounds. The gap is particularly relevant while trading is paused, because the same settlement path used for ADL is unavailable while oracle updates remain possible.

Initial deployment has a second path around the same safety assumption. `initialize()` calls `_requireAdlReplacementCapacityInvariant(..., false)`. When a nonzero 18-decimal oracle does not implement the non-standard `maxUpdateBps()` getter, the failed static call returns without running P57/P58. The stricter replacement path passes `true` and rejects that same oracle. A test oracle without the getter and an evaluator with deliberately unsafe 50% market-take bounds therefore passed initialization-style validation but reverted with P54 under replacement-style validation.

If deployment always uses a bounded `BackendPriceOracle` and the permissioned updater never publishes another round until risk processing for the preceding round is complete, off-chain sequencing supplies the missing invariant. That sequencing is not enforced or observable as an on-chain precondition, so an honest deployment error, fast series of legitimate price updates, or update during an emergency pause can exceed the movement used by the solvency proof.

Recommendation: Require a valid positive oracle movement bound during initialization whenever nonzero oracle and evaluator addresses are supplied. Enforce the bound over the cumulative movement since the last price round acknowledged by Perpetual's liquidation/ADL process, rather than independently for every backend update. For example, store an acknowledged round and reference price and prevent a further oracle update until the market has processed the current round. Also prevent price progression while the risk-processing path is paused, or introduce a separate emergency process that safely advances both together.

MX: Fixed in [6cff9da](#) with a witnessed-bankruptcy trading gate. Anyone can flag an account whose equity is negative beyond the configured dust floor and clear it after recovery. The contract stores an unbounded account mapping plus an exact count; while the count is nonzero, P87 rejects any batch containing an ordinary fill. Forced liquidation and automatic deleveraging fills remain available because they are the mechanism that repairs the insolvency. Flag and clear batches are capped at 64 entries and skip ineligible entries rather than reverting the whole batch.

This design closes the interval identified by the finding without requiring a price updater to prove that off-chain risk processing completed between every oracle round. It deliberately does not stop the cure. It also avoids a bounded queue, whose overflow could have reopened trading while additional bankrupt accounts still existed.

The accepted residual is that the gate is witness-armed: an insolvent account that nobody flags is not gated. In normal operation, the exchange witnesses the bankruptcies produced by its own oracle updates, and a permissionless, chain-fed independent watcher is specified as follow-on hardening without a delivery commitment. The contract also retains an owner escape hatch for emergency recovery.

Researcher: Partially fixed. Initialization now requires a bounded oracle, but ordinary trades remain possible before `flagBankrupt`, and cumulative oracle movement remains unbounded.

6.2.8 Global system controllers can drain or commandeer unrelated accounts

Context: [MarginBank.sol#L296-L312](#), [MarginBank.sol#L636-L674](#), [AccountManager.sol#L75-L96](#)

Description: `MarginBank` and `AccountManager` model a system controller as a single global boolean. Neither contract records which controller owns or created a particular system account.

In `MarginBank`, every approved `systemAccountController` can register or unregister any address through `setSystemAccount()`. The same controller can then transfer from or externally withdraw the balance of any address whose global `systemAccounts` bit is set. The withdrawal path checks only that the source is a system account; it never checks that the caller controls that source.

This permits two concrete privilege-boundary violations:

1. If controller A creates and funds a vault child account, independently approved controller B can call `withdrawFromSystemAccount(AChild, attacker, balance)` and drain it. Controller isolation therefore disappears as soon as more than one vault or controller is approved.
2. Controller B does not even need the victim to have been registered by another controller. It can call `setSystemAccount(victim, true)` for an ordinary `MarginBank` user and then withdraw the victim's entire bank balance to an arbitrary destination. With withdrawals paused, it can instead stage the balance into another system account and complete the exit when withdrawals resume.

`AccountManager` has the equivalent authorization problem. An approved `systemSubAccountController` can call `setSystemSubAccount(victim, attacker, true)` for any user; there is no requirement that `victim` be a system account or belong to the calling controller. The resulting bit is accepted by every consuming `Perpetual` for caller checks and margin/leverage signatures, and by `IsolatedTrader` for attacker-signed orders whose declared maker is the victim. A compromised controller can therefore commandeer unrelated users' trading accounts across every market sharing the `AccountManager`.

The precondition is that the owner has approved the malicious, compromised, or incorrectly addressed controller. Approving the `LiquidityVault` as a controller is, however, part of normal vault operation, so this creates a standing protocol-wide capability in a component that only needs authority over its own main and child accounts. The intended narrower boundary is supported by the interfaces' references to "vault-owned child accounts" and an account's controller, and by `SystemAccountHolder.controller`, which records the controller but is never consulted by either registry.

If every approved system controller is intentionally trusted as a protocol-wide custodian with authority to withdraw every `MarginBank` user's collateral and grant trading permission for every `AccountManager` user, this behavior is by design and should instead be documented explicitly.

Recommendation: Bind every system account to exactly one controller, for example with `controllerOf[account]`, and require the caller to match that controller when registering, transferring from, withdrawing from, or changing subaccount permissions for the account. During registration, verify that the target is a deployed `SystemAccountHolder` whose immutable `controller()` equals the caller, or use an owner-managed registry populated atomically when the holder is deployed.

Restrict system-account transfers to accounts under the same controller unless the source controller explicitly authorizes a cross-controller transfer. Apply the same ownership check in `AccountManager.setSystemSubAccount()`. If an owner-only recovery override is required, expose it as a separate, clearly named emergency function with dedicated events rather than giving every ordinary controller global authority. Existing deployments should migrate and verify controller bindings for all registered system accounts before re-enabling controller operations.

MXN: Fixed in [9f64c25](#).

MarginBank and AccountManager now bind each system account to exactly one controller. Registration verifies the target contract's declared controller, and every system withdrawal, transfer, and system-subaccount permission update requires the caller to match the stored binding. Cross-controller operations and attempts to register ordinary user accounts therefore fail.

Existing system accounts are migrated through inventory-checked upgrade initializers that reject missing, duplicate, mismatched, or unowned entries before controller operations become available. Tests cover two independent controllers, victim-user takeover attempts, cross-controller transfers, ownership checks, and upgrade-state preservation.

Researcher: Verified fix.

6.2.9 Binary-market oracle prices are not constrained to the payout range

Context: [Perpetual.sol#L1291-L1322](#), [Perpetual.sol#L1783-L1822](#), [Perpetual.sol#L1827-L1872](#), [PositionBalanceMath.sol#L74-L89](#), [PositionBalanceMath.sol#L127-L170](#), [BackendPriceOracle.sol#L102-L123](#)

Description: Binary-market collateral rules assume the outcome price is in the raw e18 interval $(0, 1e18]$. Perpetual identifies a binary market from Evaluator and FundingOracle configuration, but `_requireSupportedOraclePrice()` only checks that a live answer is positive and fits in `uint128`. BackendPriceOracle also has no absolute binary bound, so repeated individually capped updates can cross `1e18`.

Out-of-domain answers are then used by funding-adjusted equity, vault NAV, and leveraged-position health. A 1x binary position with its documented max-loss collateral is always reported as 100% healthy and excluded from ADL without consulting price. That shortcut is safe only while the payout domain is respected.

Most importantly, ordinary `delistPerpetual()` can freeze an oracle value above `1e18` when the owner supplies permissive bounds. Only `delistPerpetualAtSettlementPrice()` explicitly rejects a binary settlement price above one. For a one-unit 1x short opened at 0.5 with 0.5 margin, settling at 1.1 produces equity of $0.5 + 0.5 - 1.1 = -0.1$, exceeding the maximum loss for which the short was collateralized, while the pre-delist health shortcut treated it as fully safe.

This is distinct from changing whether the market is classified as binary: the configuration can remain binary throughout while the live price leaves its payout domain.

Recommendation: Enforce $(0, 1e18]$ on every oracle answer consumed by a binary market, including ordinary delisting. Where possible, configure immutable market-specific absolute bounds in the oracle itself. Do not use the fully collateralized health shortcut or ADL exclusion when the answer falls outside the domain it assumes.

MXN: Fixed at the Perpetual consumption boundary in [9d44dec](#).

Researcher: Verified fix.

6.2.10 Funding-input transitions retroactively reprice or erase already elapsed funding

Context: [Perpetual.sol#L1372-L1381](#), [Perpetual.sol#L1424-L1428](#), [Perpetual.sol#L1469-L1473](#), [Perpetual.sol#L1604-L1659](#), [Perpetual.sol#L2467-L2485](#), [FundingOracle.sol#L398-L415](#)

Description: The global funding index applies the funding rate and oracle price read at update time across the entire interval since `globalIndex.timestamp`. Changing either input must therefore first checkpoint the elapsed interval under the old regime.

`setOracle()`, `setFunder()`, and `stopTrading()` do not do this. `stopTrading()` immediately calls `FundingOracle.stopFunding()`, which resets the current rate to zero. If the index is one hour behind, the next update applies zero to that already elapsed hour and permanently erases the payment that one side owed the other. Replacing the funder can instead apply the replacement's current rate retroactively across time when the old funder was active. Replacing the oracle uses the replacement answer to price the entire old interval.

For example, assume a +0.1% hourly rate, price 100, and a global index last updated one hour ago. Calling `stopTrading()` before

checkpointing deletes a 0.1-token payment per one-unit position. No malicious owner is required: an ordinary emergency stop or component rotation while the index trails wall time is sufficient.

The intended transition pattern already exists in `setFundingRate()` and `setOffChainFundingRate()`, both of which call `_loadContext(true)` before changing the rate.

Recommendation: Advance the global index using the old oracle and funder immediately before replacing either dependency or stopping funding. If an emergency makes an old input unusable, expose a deliberate transition function with an explicit cutoff time, price, and rate rather than silently applying zero or replacement data retroactively.

MNX: Fixed in [fbc9655](#).

Researcher: Verified fix.

6.2.11 Cached index time allows expired orders and assigns funding to the wrong position holder

Context: [Perpetual.sol#L541-L632](#), [Perpetual.sol#L790-L796](#), [Perpetual.sol#L2359-L2414](#), [IsolatedTrader.sol#L957-L974](#), [FundingOracle.sol#L181-L217](#)

Description: To avoid MegaETH's volatile-data cost in the hot path, `trade()` does not advance the global index or read current block time. It uses `globalIndex.timestamp` as the batch's settlement time, checks oracle freshness relative to that same cached time, and passes it to both `IsolatedTrader` and `FundingOracle`.

If the updater stalls, the timestamp can lag wall clock without an on-chain age limit. This has several consequences:

- An order that expired after the cached timestamp still passes `order.expiration >= tradeTimestamp`, even if it is long expired in real time.
- A position closed during the gap is stored as flat at the old funding index and never pays funding for the gap. A position opened just before the eventual update is stamped with the old index and pays for the entire interval before it existed. Increases and reductions similarly apply the final quantity to the whole gap.
- Oracle staleness is checked against the old timestamp, so an oracle that is stale relative to wall clock can still be accepted.
- A newly observed trade premium is sent to `FundingOracle` with the old timestamp and can be weighted over time before the trade actually occurred.

`maxBatchesPerGlobalIndexUpdate` bounds successful calls, not elapsed time or the number of fills in one batch. Even when enabled, the first unused batch after a long idle period can contain arbitrarily many affected fills.

The README explicitly delegates current-time expiry filtering and updater liveness to off-chain infrastructure. This is therefore an architectural trust dependency rather than an undocumented implementation accident. It remains reportable if signed expiration, oracle freshness, and funding ownership are expected to be enforced on-chain under a compromised or unavailable backend.

Recommendation: Require the cached heartbeat to be within a strict wall-clock bound before any position-changing batch. Enforce order expiration against current time or a separately authenticated fresh timestamp. For correct funding ownership, checkpoint the global index before changing quantities or implement time-weighted index baselines for position-size changes.

MNX: Acknowledged. We are retaining the cached settlement timestamp on the trade hot path. Reading current block context there moves a MegaETH transaction onto a lower compute ceiling and materially reduces batch capacity. The composed fix makes that dependency explicit and bounded: the runtime rejects wall-clock-expired orders and stale oracle batches before sealing; every accepted on-chain batch consumes a configured global-index budget; and exhausting that nonzero budget forces a current-time global-index update before more trades can execute.

That update validates complete, fresh, non-future oracle metadata and resets the budget. The contract-side acknowledgement and budget behavior are in [fbc9655](#).

Researcher: Acknowledged.

6.2.12

Signed stop-order trigger prices are never enforced

Context: [IsolatedTrader.sol#L114-L130](#), [IsolatedTrader.sol#L195-L209](#), [IsolatedTrader.sol#L801-L834](#), [IsolatedTrader.sol#L897-L922](#)

Description: `Order.triggerPrice` is documented as the stop-order price and is included in the EIP-712 hash signed by the user. It is never read during order execution. Repository-wide, its only uses are the struct declaration and type hash.

As a result, the signature commits to a value that has no effect. A user can sign a buy stop intended to activate only when the oracle reaches 120, but a settlement operator can fill it while the oracle is 100 as long as the ordinary fill-price, quantity, and margin checks pass. A stop-loss can likewise be executed prematurely, crystallizing a loss or closing a position before the condition authorized by the user.

The current order schema also lacks a field distinguishing stop-loss from take-profit trigger direction. A non-zero trigger therefore cannot be enforced unambiguously without defining that semantic as part of the signed message.

This requires a user to submit a stop order and a malicious, compromised, or buggy settlement backend to activate it early. If MNX does not support stop orders at all, the field is instead misleading dead signed data and should not be exposed to clients.

Recommendation: Add an explicit signed trigger-condition type and enforce the trigger against the trusted oracle before filling; reserve zero for an order with no trigger. If triggers are intentionally enforced only off-chain, remove `triggerPrice` from the on-chain order model and clearly state that it is not a user-protecting on-chain condition.

MNX: Fixed in [6cff9da](#).

The signed order now includes an explicit trigger condition: none, price at or above, or price at or below. The contract validates the condition/price shape, rejects reserved values, and enforces the condition before any fill. Forced liquidation orders cannot carry user trigger conditions. The order schema and signing domain moved to version 2, and existing proxies migrated atomically only after a domain-hash preflight.

Two intentional behaviors are important. Activation uses the trusted oracle price, not a separately calculated mark price. Also, when a valid conditional order reaches settlement before its condition is met, it rests for retry instead of being cancelled or blamed on either account. The tests cover both directions, equality, malformed shapes, forced orders, version-1 rejection, batch freshness, and fill-price bounds.

Researcher: Verified fix.

6.2.13

Maker-side market orders bypass their own market-take bound

Context: [IsolatedTrader.sol#L384-L409](#), [IsolatedTrader.sol#L982-L999](#), [Evaluator.sol#L304-L323](#), [Perpetual.sol#L790-L796](#)

Description: A signed market order has `price == 0`. After verifying its signature, `IsolatedTrader` replaces that zero with the chosen fill price, making the normal limit-price check automatically pass. This substitution is deliberately performed for both `orderA` and `orderB`, because the batch auction may assign a market order to either side.

However, the contract performs only one market-take-bound check and always uses `orderB`'s direction. This protects a market order when it is the taker, but gives a market order assigned to `orderA` no direction-correct oracle bound.

For example, with oracle price 100 and 10% long/short bounds, Alice signs a buy market order and is assigned to `orderA`. A sell order is assigned to `orderB`, and a compromised settlement operator chooses a fill of 200. The only market bound checked is the sell-side lower bound, `fill >= 90`, so the fill succeeds. Alice's required buy-side upper bound, `fill <= 110`, is never evaluated. At 1x, Alice supplies 200 margin and can later lose 100 when price returns to the 100 oracle, transferring that value to the counterparty.

The same extreme clearing price is passed to `FundingOracle.recordTrade()`, so it can also distort funding for unrelated positions.

Recommendation: Preserve whether each original signed order was a market order and apply the direction-correct market-take bound to every market order, regardless of its maker/taker assignment. Alternatively, require every clearing price to remain inside both sides of the oracle envelope. Apply the same protection to dual forced-liquidation market orders.

MX: Fixed in [37fb2fd](#). `IsolatedTrader` now remembers whether order A was a signed zero-price market order before substituting the fill price, then applies the order-specific buy/sell market-take bound while preserving the existing order-B check. Dual market-order pairs, including dual forced-liquidation pairs, receive both directional checks. An ordinary order-A limit order is not reclassified and remains governed by its signed limit; the historical order-B bound is preserved. The contract tests and shared raw-unit parity fixture cover both directions and exact boundary values. The matching pair-aware planner/pre-seal and settlement rechecks live in the exchange monorepo and must ship before or with the contract upgrade.

Researcher: Verified fix.

6.2.14

Below-IMR accounts can increase leverage and exposure while P35 treats them as no riskier

Context: [Perpetual.sol#L1069-L1149](#), [Perpetual.sol#L2638-L2705](#), [IsolatedTrader.sol#L619-L641](#), [Evaluator.sol#L377-L395](#)

Description: When an existing account is below the initial-margin requirement, Perpetual does not require a risk-increasing update to restore the account to IMR. P35 only requires the final margin ratio to be greater than or equal to the old ratio, and P36 permits quantity growth whenever the final ratio remains above maintenance margin.

For example, assume IMR is 10% and MMR is 5%. A 10x long with quantity 10, `oiOpen` = 1,000, and margin 100 has a 6.25% margin ratio when the oracle is 96. The account can first authorize `adjustLeverage(16x)`. `MarginMath.getTargetMargin()` returns the same 100-token margin, so the economic ratio does not change and P35/P36 accept the update. The account can then add another 10 units at 96. At 16x the new exposure requires only 60 additional margin, leaving quantity 20, `oiOpen` = 1,960, margin 160, and the same 6.25% margin ratio. Absolute exposure doubled even though the account never met the 10% entry requirement.

The open-notional check may not stop the transition. `Evaluator` interprets a zero cap as unlimited, including leverage tiers that were never configured. The code comments say higher tiers may instead be blocked off-chain, making the on-chain risk limit depend on the settlement backend. If every leverage above $1 / \text{IMR}$ is disabled by a nonzero on-chain cap, this specific path is unavailable.

The caller still needs the user's signed leverage authorization and signed order, must supply proportional collateral, and a settlement operator must submit both actions. The issue is not an arithmetic error: P35 deliberately treats an unchanged ratio as no worse. It is reportable if IMR is also intended to be the maximum leverage and the on-chain layer must reject colluding or compromised-backend risk increases; otherwise it is a design-risk confirmation.

Recommendation: Require every leverage increase or quantity increase to satisfy IMR in the final state, or separately require the incremental exposure to be collateralized at IMR. Enforce an explicit maximum leverage of $1 / \text{initialMarginRequired}$, and treat absent or zero leverage-tier caps as disabled rather than unlimited unless uncapped behavior is deliberately configured through a separate flag.

MX: Fixed across the shared risk primitives in [17a7084](#) and their market integration in [6cff9da](#).

Perpetual now applies P68 whenever a transition increases position quantity or leverage. The final account must meet the full initial-margin requirement—the minimum collateral percentage for opening or increasing risk—rather than merely avoid a lower margin ratio. Risk-reducing transitions keep their existing wind-down behavior. `Evaluator` leverage tiers also fail closed for a risk increase when the required tier has not been configured; a zero cap no longer silently means unlimited exposure on that path.

The implementation is checked against a shared transition matrix covering longs, shorts, increases, reductions, direction flips, leverage changes, and boundary rounding, with contract/runtime parity fixtures.

Researcher: Verified fix.

6.2.15

Margin and leverage authorizations never expire and cannot be cancelled

Context: [SignatureVerifier.sol#L15-L31](#), [Perpetual.sol#L333-L334](#), [Perpetual.sol#L838-L985](#), [Perpetual.sol#L1069-L1123](#)

Description: The EIP-712 messages used by `addMargin`, `removeMargin`, `removeMarginUpTo`, and `adjustLeverage` contain a nonce but no deadline. The nonce is stored in a mapping of individually used values rather than as a sequential nonce that the account can advance, and the account has no function for cancelling an unused nonce.

An unused signature therefore remains valid indefinitely. The account cannot independently invalidate it on-chain; it stops being usable only if a settlement operator submits another authorization using the exact same nonce.

This is especially dangerous for `adjustLeverage`. The signature authorizes only a leverage value, while the amount of collateral moved is calculated later from the account's live position, funding state, and oracle price. For example, assume unchanged entry and market prices and a 10x position with 1,000 tokens of notional and 100 tokens of margin. A signed adjustment to 1x would add roughly 900 tokens of margin. If the request is withheld until the position has grown to 100,000 tokens of notional with 10,000 tokens of margin, the same signature can add roughly 90,000 tokens. The authorization never committed to that larger transfer.

The caller must be a whitelisted settlement operator, so an arbitrary external account cannot exploit this directly. The relevant threats are a malicious or compromised operator, a backend retaining queued signatures, or an old request being retried after substantial market and account-state changes. The fixed-amount margin operations are also exposed to execution at an unintended time, although the state-dependent leverage operation has the clearest financial impact.

This issue is separate from the MinimalForwarder deadline finding because these are Perpetual-specific EIP-712 authorizations with their own schemas and nonce storage. Expiring a forwarded meta-transaction does not expire these signatures.

Recommendation: Add a deadline to every margin and leverage authorization schema and reject execution after that deadline. Also provide an account-controlled cancellation mechanism, such as a monotonic minimum nonce that the account can advance or a function that invalidates selected nonces. Because `adjustLeverage` calculates a live transfer amount, consider additionally signing a maximum collateral-in amount, minimum collateral-out amount, or position-state version. Update the EIP-712 type hashes and domain version as appropriate.

MXN: Fixed in [6c999da](#).

Every margin, leverage, and authorized withdrawal message now signs a deadline, and the contract rejects it after that deadline. Each account can advance a minimum accepted nonce through `invalidateMarginNoncesUpTo`, cancelling every older unused authorization. Owner cancellation also invalidates subaccount-signed requests, and the cancellation call remains available while the market is stopped or delisted.

The leverage message additionally signs maximum collateral-in and minimum collateral-out bounds, so a withheld request cannot later move an unlimited amount merely because the live position grew. These schema changes use the new signature domain and were migrated atomically during the proxy upgrade. Tests cover deadline boundaries, direct and relayed cancellation, owner/subaccount interactions, replay, and collateral bounds.

Researcher: Verified fix.

6.2.16

Insolvent delist settlement is first-come-first-served

Context: [MarginMath.sol#L28-L60](#), [Perpetual.sol#L1239-L1264](#)

Description: When a market is delisted, each account should receive its remaining margin plus or minus its profit or loss at the delisting price. Before calculating that amount, `_closeSinglePositionAtDelistingPrice()` derives the collateral available to users with a checked subtraction:

```
uint128 perpBalance =  
    marginBank.getAccountBankBalance(address(this)) - feePoolAccruedAmt;
```

This assumes the Perpetual's remaining bank balance is always at least its accrued-fee reserve. That invariant can fail after insolvency, including when an unpaid funding debit is forgiven while the opposite account receives a full credit. If the bank balance falls below `feePoolAccruedAmt`, the subtraction underflows and panics before any position can be cleared. Both user-driven `closePosition()` and owner-driven `closePositionForAccounts()` then revert for every remaining account. Calling `transferAccruedFee()` cannot recover the market because it also requires the Perpetual to hold the full accrued amount. Settlement remains blocked until the market is recapitalized or upgraded.

Even when the subtraction succeeds, the remaining settlement logic distributes an insufficient balance unfairly. `getMarginLeft()` caps the current account's payment at the entire balance available after fees, and that payment is immediately transferred out. Every settlement therefore reduces the balance used to cap the next account's payment, assigning losses according to transaction order instead of applying a common haircut.

For example, if the Perpetual has five tokens remaining but has accrued ten tokens of fees, every delist close panics and no user can settle. In the less severe case where the fee reserve is covered, assume Alice and Bob are each entitled to 60 tokens but only 100 tokens remain available to users. If Alice settles first, she receives 60 and Bob receives only the remaining 40. If Bob settles first, Bob receives 60 and Alice receives 40. The owner-controlled batch function has the same behavior because it processes the supplied array sequentially.

No malicious protocol role is required for either outcome. Insolvency can arise from genuine bad debt, such as a price gap at delisting or funding forgiveness that leaves credited accounts insufficiently backed. The code does not itself create every source of insolvency, but it mishandles the resulting state in two ways: a sufficiently deep shortfall blocks all settlement, while a smaller shortfall creates a race and lets transaction or batch order determine who absorbs the loss.

Recommendation: Define an explicit insolvency waterfall between the accrued-fee claim and user settlement claims. Avoid reverting when the fee reserve exceeds the remaining balance, but do not treat a saturating subtraction alone as a complete fix: clearing users for zero while leaving an unpayable fee claim would merely hide the insolvency. Determine total collateral, recoverable fees, and total positive settlement claims before paying individual accounts, then support partial fee recovery according to the chosen priority and apply the same pro-rata haircut to every user claim. This can be implemented as a staged settlement process that records claims and fixes a global payout ratio, or through a verifiable claim snapshot. Alternatively, require an insurance fund or another backstop to recapitalize the Perpetual fully before settlement begins.

MNX: The settlement mechanism has now changed in [18432ee9](#), with final contract hardening in [c48d8599](#), verification tooling in [0fb42ee3](#), and regression coverage in [02329a38](#).

Researcher: Partially fixed: registered claims are pro-rata and underflow is resolved, but omitted claimants can still receive zero; no-registration settlements remain sequential.

Note: The initial fix here [d90da0b](#) was pushed back as incomplete.

6.2.17

Zero-supply deposits let the first depositor capture existing vault value

Context: [LiquidityVault.sol#L419-L445](#), [LiquidityVault.sol#L640-L674](#)

Description: When `totalShares == 0`, `deposit()` skips `totalNAV()` and mints one share for each deposited token. It does not verify that the vault's existing NAV is also zero.

A zero share supply does not necessarily mean that the vault owns nothing. The main or child accounts may contain residual collateral, a child may still hold a position, or value may arrive after the previous share supply reached zero through a later settlement or transfer.

For example, suppose `totalShares` is zero but the vault already owns 20 tokens of residual collateral. A user deposits 10 tokens. The zero-supply branch gives the user 10 shares, while the vault now owns 30 tokens. Because the depositor owns every share, their 10-token deposit is immediately worth 30 tokens and they capture the pre-existing 20 tokens without paying for them.

The opposite case is also harmful: if the vault has outstanding losses or negative position equity when supply is zero, the first depositor can unknowingly recapitalize those losses.

Recommendation: Before using the 1:1 initial-share branch, calculate the vault's signed economic NAV and require it to be exactly zero. Prevent child trading and position inheritance while no shares exist unless the resulting value is explicitly assigned. If residual assets or liabilities are expected, use a controlled epoch-reset process that first settles all positions and allocates the remaining value by minting shares to the intended protocol or recovery account before accepting new LP deposits.

MXN: We agree that zero share supply does not prove the vault has zero economic value, and that blindly minting one share per token can transfer residual assets or liabilities to the next depositor. In practice that state is not reachable by an outside party: the exchange itself is the vault's first depositor — each vault is seeded with the exchange's own deposit before outside deposits are possible, that deposit is already in place and remains — so no outside depositor ever interacts with the vault at zero share supply. The depositor allowlist, enabled in current operation, additionally prevents any race even at bootstrap.

The contract-level treatment depends on the pending vault share-pricing design, including how builder-vault entries, residual positions, and an epoch reset are represented. We are deliberately not adding an isolated zero-supply rule that could conflict with that model; the remainder is deferred with the vault redesign, and the redesign will ship before LiquidityVault deposits are opened to the public.

Researcher: Acknowledged.

6.2.18

Child traders can open untracked positions and corrupt vault NAV

Context: [LiquidityVault.sol#L702-L725](#), [IsolatedTrader.sol#L820-L829](#), [LiquidityVault.sol#L359-L395](#), [LiquidityVault.sol#L640-L683](#)

Description: When a child vault is created, its trader is registered as a subaccount operator in the shared `AccountManager`. This permission is global: the trader's signature is accepted by every `Perpetual` that uses that `AccountManager`, not only by the markets that the LiquidityVault owner registered and activated for the child.

However, `totalNAV()` includes position equity only from `activeChildMarkets[child]`. A trader can therefore open a position in another market without that position being counted in the vault's NAV. The same problem occurs if the owner deactivates a child/market pair while the child still has an open position, because deactivation immediately makes `totalNAV()` skip that position without checking that it has been closed.

For example, assume the vault has 100 shares and 100 tokens of real value: 50 tokens in the main account and 50 tokens in a child account. The trader moves 40 tokens of the child's collateral into a position in an untracked market. The vault still owns 100 tokens of value, but `totalNAV()` sees only the 50 tokens in the main account and the child's remaining 10-token bank balance, reporting a NAV of 60.

A second LP depositing 60 tokens consequently receives $60 * 100 / 60 = 100$ shares instead of the 60 shares they should

receive against the true 100-token NAV. After the untracked position closes and its 40 tokens return to the child's visible bank balance, total assets are 160 tokens and the second LP's 100 out of 200 shares are worth 80 tokens. They deposited only 60 tokens, so 20 tokens have been transferred from the original LPs.

Depending on the omitted position's profit or loss, the NAV can be understated or overstated, allowing incorrectly priced deposits and withdrawals.

Recommendation: Scope child-trader permissions to specific approved markets, either through market-specific authorization in `AccountManager` or by routing child trades through a vault-controlled function that checks the market allowlist. Separate "new trading allowed" from "included in NAV": disabling a market should prevent new exposure but continue including its existing position in NAV until that position is fully closed. Before finally removing a market from NAV tracking, verify on-chain that the child has no remaining position in it.

MXN: Fixed in [9f64c25](#).

System-account operator authority is now keyed by account, operator, and market with three states: none, reduce-only, and active. Perpetual consults that market-scoped permission for orders and account operations. A vault can move a child/market pair to reduce-only to stop new exposure while continuing to include the position in net asset value, and it removes the pair from valuation only after the position is flat. Existing permissions are migrated through a one-time, inventory-checked initializer that fails closed until the legacy set is accounted for.

The regression suite covers cross-market rejection, reduce-only exits, deactivation with an open position, migration completeness, and the final flat position removal.

Researcher: Verified fix.

6.2.19 `getMarginRatio()` reports a worthless long as fully healthy when notional rounds to zero

Context: [PositionBalanceMath.sol#L111](#)

Description: The function `getMarginRatio()` in `PositionBalanceMath` returns `BaseMath.baseInt()` (+100% margin ratio) when a long position's current notional rounds down to zero.

For longs, the margin ratio is effectively:

$$MR = 1 - \text{debt} / \text{currentNotional} = \text{oiOpen} - \text{margin} / \text{currentNotional} = \text{qPos} * \text{price} / 1e18$$

If the long is leveraged, $\text{debt} > 0$. As `currentNotional` approaches zero, the true margin ratio approaches negative infinity. Returning +100% therefore flips the result from maximally unhealthy to fully healthy.

This is reachable for leveraged binary longs. `getMarginRatioWithBinaryCheck()` only special-cases fully collateralized 1x binary positions and binary shorts; binary longs fall through to the standard `getMarginRatio()` path.

Since binary markets can legitimately trade near their minimum price, a small long can satisfy $\text{qPos} * \text{price} < 1e18$, causing `currentNotional` to round to zero.

Recommendation: In the long branch, handle `balance == 0` based on debt direction. If $\text{debt} > 0$, return a maximally unhealthy value such as `-BaseMath.baseInt()` or `type(int256).min`. Returning `+BaseMath.baseInt()` is only directionally correct when the long has no debt, or in the short branch where notional approaching zero makes the short's margin ratio tend toward positive infinity.

MXN: Fixed in [19018ea](#). `getMarginRatio()`'s long branch now returns -100% (`-BaseMath.baseInt()`) instead of +100% when the notional rounds to zero and the position has debt ($\text{oiOpen} > \text{margin}$); a debt-free long stays healthy and the short branch is unchanged. We use `-BaseMath.baseInt()` rather than `type(int256).min` so the sentinel survives downstream arithmetic. Covered by exact-value unit tests plus an integration test that drives a fractional-step binary long through the real liquidation gate.

Researcher: Verified fix.

6.2.20 `_isBinaryFutureMarket()` infers the entire margin model from three admin-settable parameters, so a config change silently flips a market's regime

Context: [PositionBalanceMath.sol#L30-L32](#)

Description: `_isBinaryFutureMarket()` derives "binary" at call time from `liveEvaluator.minPrice()`, `Evaluator.maxPrice()`, and `FundingOracle.maxFunding()` via `PositionBalanceMath.isBinaryBoundedMarket`.

All three are admin-mutable. Because this predicate gates the max-loss floor, the removable-margin override, the target-margin override, ADL eligibility, and delist clamping, changing any one of `minPrice/maxPrice/maxFunding` silently switches the market's entire margin regime with no revert and no distinguishing event: a binary market that has `maxPrice` retuned loses the max-loss floor protecting its shorts, and a vanilla perp that transiently matches (1, 1e18, 0) acquires binary health math.

This is the same predicate everything hangs off, so a benign-looking parameter tweak becomes a solvency change.

Recommendation: Set an explicit, immutable `isBinary` flag at market creation and gate all of the above on it, rather than inferring type from mutable price/funding config. If inference must stay, make the type transition an explicit, guarded, event-emitting operation and forbid changing the fingerprint parameters while positions are open.

MX: Fixed in [3411011](#). The regime is now a pinned `isBinaryFutureMarketFlag` set at `initialize()`, with a one-time owner-gated migration for already-deployed markets (`initializeBinaryFutureMarketFlag`, `reinitializer(3)`; P63 makes it single-use, P66 orders it after the delist migration) and a fallback to the old inference until migration runs. Flagged binary markets get one-way locks (`Evaluator.lockBinaryMarketPriceBounds`, `FundingOracle.lockBinaryMarketFunding`), and the price/funding setters plus `setEvaluator/setFunder` reject configs that would break a locked market (P64/P65). Once pinned, no single admin change can flip the regime, widen a binary band beyond [1 wei, 1e18], or enable funding on a binary market. Covered by `BinaryMarketRegime.test.ts`.

Researcher: Verified fix.

6.2.21 Forwarded meta-transactions have no deadline, enabling stale signatures to be executed against changed state

Context: [MinimalForwarder.sol#L14](#)

Description: The `ForwardRequest` struct in `MinimalForwarder.sol` has no expiry field, and `verify()` checks only the signer and `req.nonce == nonces[req.from]`.

A signed gasless request is therefore executable for unbounded wall-clock time until the signer's single sequential nonce is advanced by landing some request.

A signer cannot time-box an action or cheaply retire a pending signature. This lets a relayer spring a withheld request at a moment of its choosing, or when several requests share a nonce because an earlier one stalled choose to land a stale one over the fresh one.

The sequential nonce is only a partial mitigation: a stale signature dies once the signer lands any other gasless action, so the exposed cases are a withheld request during a quiet period, a malicious/compromised relayer, or state that changes without touching the forwarder nonce (e.g. a direct ERC-20 approve).

Because several value-moving and authority-granting entrypoints trust this forwarder and read their effect from mutable state rather than from the signed calldata, the missing deadline has concrete downstream impact.

There are three realized impact instances, all fixed by the one change:

1. **MarginBank.depositFullAllowanceToBank sweeps a future allowance.** The call binds only `_account`; the amount is read live from ERC-20 allowance via `msgSender()`. A withheld signature executed after the user raises their allowance (a direct

approve, which doesn't bump the forwarder nonce) deposits the full new allowance an amount and time-unbounded authorization pulling capital into the exchange at an unintended time.

2. **AccountManager.setSubAccount becomes a delayed account-takeover window.** The relay-enabled grant is honored across the trading stack - `removeMargin/removeMarginUpTo/addMargin/adjustLeverage/closePosition` and order signing as maker.

A withheld grant can be sprung far later (e.g. after a once-trusted session key is compromised, or after the user assumes the grant never landed), yielding full operational control of the account's capital until an explicit `revoke.msgSender()` and binds a rate but not a time.

3. **Perpetual.setOffChainFundingRate replayed in the wrong market state.** This privileged operator call is relay-enabled and binds a rate but not a time. A withheld or same-nonce-competing signature can install an outdated funding rate after the market has moved.

Recommendation: Add a `uint256` deadline to `ForwardRequest`, include it in `TYPEHASH`, and enforce `require(block.timestamp <= req.deadline, 'MinimalForwarder: expired')` in `verify()` and `execute()` functions.

MNX: Fixed in [860315a](#). `ForwardRequest` and the EIP-712 typehash now include a deadline; both `verify()` and `execute()` reject `block.timestamp > deadline`, ending unbounded-lifetime signatures. EIP-712 domain version bumped to 0.0.2. Together with the new relayer allowlist and the nonce rollback in issue with title: `execute()` increments the nonce before the target call and swallows target reverts, this closes the withheld-signature scenarios.

Researcher: Verified fix.

6.2.22

Signature-authorized withdrawals bypass the direct-withdrawal timelock

Context: [MarginBank.sol#L565](#)

Description: `MarginBank` exposes two owner-authorized exit paths with unequal protection.

The direct path (`requestWithdrawFromBank` → `executeWithdrawFromBank`) enforces a delay (`directWithdrawalDelaySeconds`, floored to 60s) and creates a cancellable request, giving a per-withdrawal reaction/dispute window.

The signature path (`withdrawFromBankWithAuth`) validates an EIP-712 owner signature, nonce, and deadline, then calls `_executeWithdrawal` immediately with no delay, no cancellable window.

Both paths share only the guardian's coarse global halt (`sWithdrawalAllowed()`). Because the same owner key can exit through either route, the timelock is fully bypassable: an attacker holding a valid owner signature (from a compromised key, a phished signature, or a malicious relayer sitting on a signed request with a distant deadline) drains funds instantly and irreversibly, with no chance for the owner to cancel and effectively no window for the guardian to react to that specific withdrawal.

If the direct-path timelock is intended as a security/reaction control, this asymmetry defeats it; note also that `withdrawFromBankWithAuth` is permissionless, so any holder of a valid signature not only a trusted operator can submit it.

Recommendation: Clarify the intended purpose of the direct-withdrawal timelock. If it is a security control, apply an equivalent delay/cancellable window (or a mandatory signed-withdrawal delay) to `withdrawFromBankWithAuth` so both paths offer the same protection; otherwise, document why instant signature-authorized withdrawals are acceptable and specify the compensating control (e.g., operator gatekeeping, enforced short deadline bounds) that bounds the exposure.

MNX: Fixed in [33bd129](#) and completed by [6cff9da](#). `withdrawFromBankWithAuth` is now gated by an owner-managed withdrawal-Relayers allowlist checked against `msg.sender` (not `msgSender()`), so it cannot be satisfied through the trusted forwarder. The timelocked direct-withdrawal path remains the permissionless escape hatch; `NatSpec` documents the two-path split.

Researcher: Verified fix.

6.2.23

Withdrawal requests never expire, turning the timelock into a standing redemption option

Context: [LiquidityVault.sol#L512](#)

Description: `executeWithdrawal` enforces only that a request has matured (`block.timestamp * 1000 >= request.unlockAtMs`) with no upper time bound, and `requestWithdrawal` does not burn the shares (they are burned only at execution).

An LP can therefore pre-arm a withdrawal, let the timelock elapse, and hold the matured request indefinitely, exercising it instantly at whatever future moment NAV is most favorable since the payout is priced at execution-time NAV.

This defeats the delay's purpose: instead of preventing NAV-timed exits, the timelock becomes a one-time cost that, once paid, grants a permanent instant-exit option able to capture NAV spikes a timelock-bound LP would miss, transferring value from passive LPs.

The single-pending-request constraint only limits each LP to one standing option over their full balance, which is sufficient to game it; extraction per exercise is bounded by the insufficient Main liquidity check.

Recommendation: Add an execution window so a request is valid only within `[unlockAtMs, unlockAtMs + executionWindowMs]` and expires (auto-cancelled, requiring a fresh timelocked re-request) afterward, sized long enough for honest execution but short enough to prevent a standing option; complement with request-time or `min(request, execute)` NAV pricing so exit value cannot be optimized against later marks.

MNX: Acknowledged. This issue is deferred until the LiquidityVault share-pricing redesign is complete. The redesign will ship before LiquidityVault deposits are opened to the public.

Researcher: Acknowledged.

6.2.24

Permit front-running grieving causes DoS in `depositToBankWithPermit`

Context: [MarginBank.sol#L363](#)

Description: `depositToBankWithPermit` calls `IERC20Permit(token).permit(...)` inline and unconditionally before executing the transfer:

```
IERC20Permit(token).permit(caller, address(this), _amount, _deadline, _v, _r, _s);
_depositFrom(caller, _account, _amount);
```

EIP-2612 permit is authenticated by the owner's signature and consumes a single-use, monotonic nonce on the token contract. The signature `(_v, _r, _s)` carried in the calldata is a public, standalone authorization: anyone who sees the pending transaction in the mempool can extract those parameters and submit `token.permit(caller, address(this), _amount, _deadline, _v, _r, _s)` directly to the token contract in a separate transaction.

When an attacker (or any unrelated actor, including an honest relayer racing the same signature) lands that bare permit call first, the owner's nonce is incremented and the allowance is set.

The victim's `depositToBankWithPermit` transaction then executes against a stale nonce, so the nested permit call reverts with an invalid-signature error. Because the permit call is inline and its revert is not handled, the entire deposit reverts even though the collateral approval it was trying to obtain now already exists on the token.

Recommendation: Do not let a failed permit revert the whole deposit. Wrap the permit call in a try/catch, and after it, verify that a sufficient allowance is in place before proceeding, so that a front-run permit (which leaves the correct allowance) results in a successful deposit rather than a revert.

MXN: Fixed in [33bd129](#). The inline permit() is now wrapped in try/catch with an allowance fallback: a front-run/consumed permit no longer reverts the deposit as long as sufficient allowance exists, while an invalid signature with no allowance still reverts clearly. The unit test simulates the front-run by consuming the permit directly before the deposit.

Researcher: Verified fix.

6.3 Low Risk

6.3.1 AccountManager's unvalidated forwarder can grant arbitrary session keys

Context: [AccountManager.sol#L45-L67](#), [BaseRelayRecipient.sol#L27-L49](#), [Perpetual.sol#L1732-L1744](#)

Description: AccountManager stores its initializer-supplied `_trustedForwarder` without checking that it is the intended Minimal-Forwarder, and exposes no function to rotate it. `setSubAccount()` attributes the call to `_msgSender()`, which trusts the final 20 calldata bytes whenever `msg.sender` equals that configured address.

If an EOA or malicious contract is mistakenly supplied, it can call `setSubAccount(attacker, true)` with a victim address appended to calldata. AccountManager records the approval under the victim even though the victim signed nothing. The attacker can repeat the operation after any attempted user revocation because the malicious forwarder remains permanently trusted.

The resulting authorization is consumed by every Perpetual sharing the AccountManager. It can authorize attacker-signed orders as the victim and satisfy subaccount checks for margin, leverage, and position-management operations. One initialization mistake therefore compromises account authority across all connected markets.

Recommendation: Validate the forwarder during initialization through an owner-managed registry, expected code hash, or deployment-factory invariant, and reject the zero address unless relay support is deliberately disabled. Add a timelocked, event-emitting rotation path so a compromised or obsolete forwarder can be revoked. A replacement should not become active until its identity and domain configuration have been checked.

MXN: Fixed in [9f64c25](#) and the on-chain code gate is in [9f64c25](#), and the canonical attestation, guarded rotation path, and regression suite are in [b812aa97](#).

Researcher: Fixed. The code gate, owner-only rotation, and canonical attestation flow adequately address the reported forwarder-misconfiguration risk. Direct bypass of the operational guard requires the trusted owner and is accepted as part of the deployment and rotation trust model.

6.3.2 Guardian ownership transfer leaves the former owner as emergency operator

Context: [Guardian.sol#L98-L118](#), [Guardian.sol#L126-L160](#), [Guardian.sol#L163-L230](#)

Description: `Guardian.initialize()` makes the initializer caller both `owner()` and `guardianOperator`. Accepting an inherited two-step ownership transfer changes only `owner()`; it does not update the independent guardian role.

After a deployment key transfers ownership to the production multisig, the deployment key can therefore still pause or re-enable trading and withdrawals for arbitrary supplied markets. Because the guardian operator also satisfies `onlyFundingStatusOperator`, it can additionally switch funding between on-chain and off-chain modes. The new owner can repair the state through `setGuardian()`, but the contract does not make that separate handoff explicit or atomic.

The same behavior survives `renounceOwnership()`: the previous owner remains the emergency operator while no owner remains able to rotate it. If the initializer is deliberately the permanent hot guardian, this separation is expected. Otherwise, an operational process that treats ownership transfer or renunciation as retiring the deployment key leaves privileged authority behind.

Recommendation: Accept explicit initial owner, guardian operator, and funding-status operator addresses during initialization rather than deriving multiple roles from the caller. Provide an atomic role-handoff operation, or make ownership acceptance re-

quire the pending guardian assignment to be finalized at the same time. Reject an accidental zero guardian, emit both previous and new role holders, and make deployment validation prove that no retired key retains authority.

MXN: Fixed in [83cefb2](#).

Researcher: Verified fix.

6.3.3 FundingOracle's guardian is unvalidated and cannot be rotated

Context: [FundingOracle.sol#L125-L161](#), [FundingOracle.sol#L327-L365](#)

Description: `FundingOracle.initialize()` accepts any `_guardian`, including the zero address or an unrelated account, and stores it in `guardianContract`. No setter can correct or rotate that address after initialization. `setFundingRateStatus()` is exclusively gated by the stored value.

Because the default enum value is `ALLOWED`, a zero or inaccessible guardian leaves on-chain funding enabled and prevents the owner from moving the market to off-chain funding during an incident. If the mistaken address is controlled by someone else, that address can instead disable on-chain funding, zero the current rate, or switch modes at an unintended time. Replacing the external Guardian contract later also leaves this oracle bound to the retired deployment.

The owner can rotate the funding-rate provider and maximum rate, but cannot repair the guardian dependency through any ordinary lifecycle operation. Recovery therefore requires a proxy upgrade or replacement rather than the same governed configuration path used for the other `FundingOracle` roles.

Recommendation: Reject the zero address during initialization and add an owner-controlled, event-emitting guardian rotation method. Prefer a two-step or handshake-based rotation that verifies the replacement Guardian recognizes this `FundingOracle` and its provider before authority changes. Deployment checks should assert that `FundingOracle`, `Guardian`, and `Perpetual` all reference the intended counterparties.

MXN: Fixed in [fbc9655](#).

Researcher: Verified fix.

6.3.4 Future-dated oracle rounds bypass the staleness check

Context: [Perpetual.sol#L1855-L1867](#), [Perpetual.sol#L1927-L1937](#), [Perpetual.sol#L2359-L2390](#)

Description: `Perpetual`'s freshness check enforces only the lower bound:

```
updatedAt + oracleMaxStalenessSeconds >= referenceTimestamp
```

It never requires `updatedAt <= referenceTimestamp` or `updatedAt <= block.timestamp`. A feed that reports a round timestamp one year in the future therefore passes immediately and continues to be considered fresh until that future time plus the configured tolerance. `_requireValidOracle()` also validates only decimals and answer range when an oracle is installed; it discards all round metadata.

The bundled `BackendPriceOracle` always stamps updates with the current block timestamp, so this does not arise through its normal update path. `Perpetual` nevertheless accepts any contract implementing the oracle interface, including during initialization and replacement. A wrong feed adapter, timestamp-unit error, or replacement oracle with malformed future metadata can consequently pin a stale or incorrect answer as fresh across trade settlement, funding updates, liquidation, ADL, and funding-adjusted account valuation.

`Perpetual` also discards `roundId` and `answeredInRound`, so it cannot reject an incomplete round using the usual aggregator metadata invariant.

The missing upper timestamp and round-completeness bounds allow malformed metadata to bypass freshness even when the reference timestamp is current and the reported price itself is within the supported numeric range.

Recommendation: Require a nonzero `updatedAt` that is no greater than the relevant current timestamp, and then compute age without addition overflow, for example `referenceTimestamp - updatedAt <= maxStaleness`. When consuming Chainlink-compatible metadata, also require a valid round and `answeredInRound >= roundId`. Apply the same validation before accepting a replacement oracle.

MNX: Fixed in [fbc9655](#) and [5caa2b8e](#).

Researcher: Verified fix.

6.3.5 Evaluator reconfiguration can eliminate the execution grid and block forced exits

Context: [Evaluator.sol#L224-L255](#), [Evaluator.sol#L445-L500](#), [IsolatedTrader.sol#L399-L404](#), [IsolatedADL.sol#L148-L158](#)

Description: Evaluator validates each mutable price and quantity parameter in isolation, but it does not verify that the resulting parameter set contains any executable grid point.

For example, with `minQty = 1e18` and the effective maximum quantity equal to `2e18`, the owner can set `stepSize = 3e18`. The setter accepts every nonzero value, but no positive multiple of `3e18` lies in the permitted `[1e18, 2e18]` range. Every call to `verifyQuantityChecks()` then reverts. The equivalent price failure is possible by setting `tickSize` above `maxPrice`, with no multiple falling between `minPrice` and `maxPrice`.

These checks are not limited to new risk. `IsolatedTrader` applies them to ordinary reductions and forced-liquidation fills, while `IsolatedADL` applies them before clamping its requested amount to the parties' live positions. An empty grid therefore blocks trading, liquidation, and ADL market-wide. Less extreme live changes can strand individual positions: raising `minQty` above an existing position size, or selecting a step that the position does not align with, prevents a reduce-only full close. Forced liquidation cannot work around that case by overfilling because it separately requires the fill quantity not to exceed the position.

The failure occurs before position accounting or proportional reduction math is reached: an internally inconsistent or live-incompatible Evaluator configuration rejects every usable input amount.

Recommendation: Validate the configuration as a set. For quantities, require that `ceil(minQty / stepSize) * stepSize` does not exceed either effective maximum. Apply the analogous test to the price range and tick size. Before activating live changes, verify compatibility with existing positions or introduce a migration process.

Independently, permit an exact full-position, strictly risk-reducing close or liquidation even when a position was created under an older minimum or step configuration. Keep that exemption narrow so it cannot open or increase exposure.

MNX: Fixed in [a3b0f5c](#).

Researcher: Verified fix.

6.3.6

Ownership transfer leaves the initial owner authorized as a price updater

Context: [BackendPriceOracle.sol#L53-L77](#), [BackendPriceOracle.sol#L102-L117](#)

Description: `BackendPriceOracle.initialize()` makes the initializer caller both the contract owner and an authorized price updater. Ownership is later transferred through inherited `Ownable2StepUpgradeable` logic, which changes the owner but does not modify the independent updaters mapping.

After a deployer transfers ownership to the production multisig, `owner()` correctly reports the multisig, but the deployer remains authorized to call `setPrice()`. The new owner is not automatically an updater either. Unless the multisig separately grants itself the updater role and revokes the deployer, a supposedly retired deployment key retains live oracle authority.

The former owner cannot change configuration after the transfer, but it can continue publishing individually capped rounds. Repeated rounds can move the answer arbitrarily over time and affect position health, liquidation, funding, settlement, and vault NAV. The same problem makes `renounceOwnership()` misleading: the former owner can remain a price updater even though all owner-only administration has supposedly been disabled.

The risk does not arise if the production multisig initializes the proxy directly or if updater revocation is performed as an explicit handoff step. The contract itself neither enforces nor atomically supports that step.

Recommendation: Make updater handoff an explicit, atomic part of ownership rotation. If the owner is always intended to be an updater, revoke the previous owner and authorize the new owner when ownership is accepted, including clearing the updater on renunciation. If updater and owner roles are intentionally independent, accept an explicit initial updater during initialization rather than silently granting the role to the initializer caller, and provide a documented rotation function or deployment invariant that proves no retired key remains authorized.

MXN: Fixed in [a4396dc](#).

Researcher: Verified fix.

6.3.7

Switching funding off-chain forces a zero rate for the rest of an already-set window

Context: [FundingOracle.sol#L222-L260](#), [FundingOracle.sol#L327-L366](#), [FundingOracle.sol#L465-L477](#)

Description: Guardian can move a market from on-chain funding computation to a whitelisted off-chain rate. After attempting to checkpoint the old rate, `FundingOracle.setFundingRateStatus(NOT_ALLOWED)` sets `currentFundingRate` to zero. It does not change `windowSet`.

If the on-chain rate was already installed for the current nominal window, the off-chain operator cannot replace it. For example, during funding window 2, `windowSet` is 1. After the status transition zeroes the current rate, `setOffChainFundingRate()` computes `expectedWindow = 2` and requires `windowSet < expectedWindow - 1`, or `1 < 1`. The call reverts with `FundingOracle: funding rate for current window is already set`.

Funding therefore remains zero until the next nominal boundary, potentially for almost one hour. This defeats the documented purpose of the off-chain setter as an override for the ongoing window and can remove up to one configured hourly funding payment from one side of all open positions. The status transition is privileged and may be rare, which limits exploitation, but an ordinary emergency move away from an incorrect on-chain rate triggers the behavior without malicious administration.

Recommendation: After Guardian checkpoints the old rate, allow exactly one mode-transition replacement for the remainder of the current window. Track whether a rate was installed under the current computation mode, or use a separate transition nonce, rather than treating the prior on-chain rate as if it were already the requested off-chain rate. If zero funding for the rest of the window is intentional emergency policy, document it explicitly and remove the interface claim that the off-chain rate overrides the ongoing window.

MXN: The fix is in [d2c92a29](#), with final hardening in [c48d8599](#) and regression coverage in [02329a38](#).

Researcher: Verified fix.

6.3.8

A missed funding update can create a one-second observation window

Context: [Perpetual.sol#L1604-L1614](#), [FundingOracle.sol#L262-L325](#), [FundingOracle.sol#L453-L487](#)

Description: `Perpetual.setFundingRate()` is permissionless. `FundingOracle` decides whether another rate can be set from the nominal hour number, but after every successful call it resets the observation window's `tFirst` to the caller's actual timestamp.

If one scheduled update is missed, these two clocks can diverge. Assume funding starts at `t0` and no one sets the first rate at the one-hour boundary. A caller can set it at `t0 + 7,199`, one second before the two-hour boundary. `FundingOracle` records `windowSet = 1` and resets `tFirst` to `t0 + 7,199`. At `t0 + 7,200`, nominal window 3 begins, so the same caller can set another rate immediately. That second calculation divides by an elapsed observation period of only one second, then installs the result for the following hour.

The PoC kept the premium at zero for 7,198 seconds and changed the last recorded premium to +10% for one second. The late catch-up rate was small because almost the full two-hour history was included. One second later, the next call treated only the +10% second as its entire sample, reached the configured 0.1% hourly maximum, and produced a per-second rate more than 1,000 times the catch-up rate. Once installed, honest callers cannot correct it within the same nominal window.

The attacker does not need a privileged role to time the two funding-rate calls and can select the opportunity after observing the last recorded premium. Deliberately choosing that premium additionally requires a trade to pass through the permissioned settlement path. Exploitation also requires the expected hourly update to have been missed for nearly a full additional window because of a keeper outage or equivalent liveness failure.

Recommendation: Base funding observations on fixed nominal start and end boundaries rather than the timestamps at which keepers happen to call. When catching up a missed window, either compute only the completed nominal interval and retain subsequent samples for the next interval, or deliberately skip the missed update without resetting `tFirst` into a nearly finished window. At minimum, require a full observation duration before another on-chain rate can be installed.

MXN: The calculation was removed in [d2c92a29](#), with final hardening in [c48d8599](#) and regression coverage in [02329a38](#).

Researcher: Verified fix.

6.3.9

Evaluator applies both quantity caps per fill, allowing oversized orders to be split

Context: [Evaluator.sol#L220-L259](#), [IsolatedTrader.sol#L396-L409](#), [IsolatedTrader.sol#L1016-L1046](#)

Description: `Evaluator` stores separate maximum quantities for limit and market orders, but `verifyMinMaxQuantityChecks()` does not receive an order type. It consequently requires every supplied quantity to be below both `maxQtyLimit` and `maxQtyMarket`. The effective cap for every order type is therefore always the smaller of the two configured values, leaving the larger setting with no effect.

`IsolatedTrader` compounds this mismatch by passing only `fill.quantity` to `verifyQuantityChecks()`. The signed `order.quantity` is never checked against either configured maximum. `_fillQuantity()` merely ensures that cumulative fills do not exceed that signed quantity, so a large order can be divided into any number of individually compliant fills.

For example, assume `maxQtyLimit = 100` and `maxQtyMarket = 10`. A user can sign a 100-unit market order and the settlement engine can execute it as ten 10-unit fills. Every fill passes both `Evaluator` comparisons and cumulative filled quantity never exceeds the signed 100 units, so the full 100-unit market order executes despite the documented 10-unit market maximum. At the same time, a 50-unit limit fill is rejected by the market-order cap even though it is below the configured limit-order maximum.

The open-notional tiers do not make these controls equivalent. They may be disabled for a leverage tier, and quantity caps can also be intended to bound execution size for position reductions and other trades that do not increase open notional. If MXN intentionally

treats these values as a single per-fill resource limit, the two separately named and documented settings should be replaced with one setting and the behavior documented.

Recommendation: Pass the order type into Evaluator and apply `maxQtyMarket` only to market orders and `maxQtyLimit` only to limit orders. Validate the signed total order quantity, or track the aggregate quantity executed for that order or batch, rather than checking each fill in isolation. Add tests in which an oversized signed order is split across several individually valid fills.

MXN: Fixed in [a3b0f5c](#).

Researcher: Verified fix.

6.3.10

Funder rotation does not validate the provider pairing or synchronize the replacement lifecycle

Context: [Perpetual.sol#L790-L796](#), [Perpetual.sol#L1420-L1473](#), [FundingOracle.sol#L125-L220](#)

Description: This finding applies when a live market's funder is replaced. If MXN always stops the market, checkpoints funding, validates the replacement, and restarts its funding schedule as one controlled migration, the issue is operationally mitigated.

`Perpetual.setFunder()` stores any supplied address without checking that it is a `FundingOracle` whose `provider()` is this `Perpetual`. It also does not require trading to be stopped or synchronize the replacement's `fundingStart`, current window, or current rate with the market.

If the replacement `FundingOracle` names a different provider, an otherwise valid regular trade performs its position and margin calculations but eventually calls `recordTrade()`. The replacement rejects that call in `onlyFundingRateProvider`, reverting the entire batch. Moving funding off-chain does not avoid the failure because the provider modifier runs before `recordTrade()` can return early.

A fresh correctly paired replacement fails differently. Its `fundingStart` is zero, so `recordTrade()` accepts calls but does not record any premium and its funding rate remains zero. Because the `Perpetual`'s existing `tradingStartTime` is still nonzero, `startTrading()` cannot start the replacement schedule. The market can therefore keep trading while funding is silently inactive until the owner deliberately stops and restarts it. Conversely, reusing an already active `FundingOracle` imports its old start time, window, and rate into the new market.

For example, suppose market `P` is already trading and the owner deploys a fresh `FundingOracle` with `provider = P`, then calls `P.setFunder(newOracle)`. Trades continue, but the `fundingStart > 0` branch in `recordTrade()` is never reached. Calling `P.setFundingRate()` cannot establish a rate because the expected funding window is zero, while calling `P.startTrading()` reverts because `P` already has a start time. Restoring normal funding requires explicit market downtime and a stop/start sequence.

These failures remain possible even if elapsed funding under the old oracle is checkpointed correctly before replacement. Provider compatibility and replacement lifecycle state must be validated independently to prevent regular trading from reverting or funding from remaining silently inactive.

Recommendation: Replace the funder through a dedicated migration function that requires trading to be stopped, checkpoints the old funder, rejects zero or non-contract replacements, and verifies `IFundingOracle(newFunder).provider() == address(this)`. Explicitly initialize or validate the replacement's funding schedule before trading resumes. Consider a two-sided handover in which both `Perpetual` and `FundingOracle` confirm their pairing, rather than allowing their independent setters to create a partially configured relationship.

MXN: Fixed in [fbc9655](#). Funder replacement is now a stopped-market handover. `Perpetual` requires the replacement to be a contract that names this `Perpetual` as its provider, and the `FundingOracle` independently confirms the pairing. The replacement must have a clean, never-started or properly stopped funding schedule; a live rate, window, or sample cannot be imported from another market. The outgoing interval is checkpointed before the address changes, composing with the *Funding-input transitions retroactively reprice or erase already elapsed funding* transition fix.

FundingOracle provider rotation validates nonzero contract code, emits previous and new addresses, and ownership renunciation is disabled so a broken pairing cannot become permanently unrepairable. Tests cover wrong providers, active schedules, clean stopped schedules, handover ordering, and proxy upgrades.

Researcher: Verified fix.

6.3.11

MarginBank direct-withdrawal requests never expire

Context: [MarginBank.sol#L499-L530](#), [MarginBank.sol#L684-L730](#), [MarginBank.sol#L733-L783](#)

Description: MarginBank checks an owner's current bank balance when a direct-withdrawal request is created, but it neither reserves that balance nor invalidates the request when the balance is later spent or transferred. Execution checks only that `block.timestamp >= executableAt`; the request has no upper validity bound.

An owner can therefore request its entire balance, let the request mature, move the original balance away through trading or another bank operation, and retain the request indefinitely. If funds later return to the account, even months afterward, the old request becomes immediately executable against those newly returned funds without a fresh delay.

The amount and destination are fixed and execution remains restricted to the request owner, which limits direct theft risk. The behavior nevertheless turns the delay into a one-time, pre-armable condition instead of a reaction window attached to the funds being withdrawn. If the direct-withdrawal delay is intended as a meaningful operational or incident-response control, an indefinitely mature request weakens it.

Recommendation: Add a bounded execution window and require an expired request to be cancelled or replaced by a newly delayed request. If the delay is meant to bind a particular balance, reserve the requested amount when the request is created or cancel/revalidate the request whenever an outgoing balance mutation makes the unreserved balance insufficient.

MXN: Fixed in [484a88d](#). Each new direct-withdrawal request stores an immutable `expiresAt` and may execute only from `executableAt` through that deadline, inclusive. An expired request stops blocking direct and relayed withdrawals; a replacement retires it through the normal cancellation event, and a stale request id cannot clear a newer active request. The atomic upgrade initializer refuses any historical request that is not already executed or cancelled and fails closed if skipped. Contract, boundary, storage-layout, and proxy-upgrade tests are included; the runtime ingest and deployment campaign remain in the exchange monorepo.

Researcher: Verified fix.

6.3.12

IsolatedTrader caches an unvalidated caller-supplied chain ID

Context: [IsolatedTrader.sol#L300-L328](#)

Description: `IsolatedTrader.initialize` accepts `_chainId` from the initializer caller and uses it to construct the cached EIP-712 domain separator without requiring it to equal `block.chainid`.

If the wrong value is supplied during deployment, orders signed using the actual network's chain ID are invalid because their domain does not match the permanently cached value. There is no setter or reinitializer that can repair the domain without upgrading or redeploying the contract.

More importantly, if the proxy address and relevant accounts are reproduced across two chains, configuring the destination contract with the source chain's ID makes orders signed for the source domain valid on the destination as well. Cancellation and fill state are chain-local, so an order consumed or cancelled on one chain may remain executable on the other. The verifying-contract address requirement narrows this replay condition but does not eliminate it for deterministic or intentionally mirrored deployments.

Because exploitation requires a deployment/configuration error or deliberate cross-chain address alignment, the issue is classified as Low.

Recommendation: Remove `_chainId` from the initializer and construct the domain from `block.chainid`. At minimum, require

`_chainId == block.chainid`. Consider OpenZeppelin's EIP-712 implementation, which also rebuilds the separator when the active chain ID changes instead of permanently caching a stale domain across a chain split.

MX: Fixed in [6c99da](#).

Researcher: Verified fix.

6.3.13 Whitelisted trader fees bypass the default 25% cap

Context: [IsolatedTrader.sol#L411-L430](#), [IsolatedTrader.sol#L492-L505](#), [Perpetual.sol#L1581-L1601](#)

Description: Perpetual caps both default maker and taker fees at 25%. `IsolatedTrader.setWhitelistedTrader`, however, accepts arbitrary `uint128` custom maker and taker fees and stores them without applying the same bound. When the whitelist entry is active, these values replace the capped defaults during trade execution.

The whitelist documentation says whitelisted traders do not pay fees, but the implementation permits any fee, including values above 100%. A test setting a 200% custom maker fee successfully produced a fee twice the order notional and a maker funds flow consisting of the position margin plus that fee.

Orders do not commit to a maximum fee. Consequently, a mistaken whitelist configuration or a fee change made after an order was signed can impose a cost far beyond the protocol's advertised default limit. Extremely large custom fees also reach shared fixed-point helpers whose `uint128` results assume economically bounded inputs.

The setter is owner-only, so this is primarily a privileged misconfiguration and signed-order expectation risk rather than an unprivileged fee-manipulation path.

Recommendation: Enforce the same maximum used by `setDefaultMakerFee` and `setDefaultTakerFee` for custom fees. If whitelisting is meant only to grant discounts, additionally require each custom fee to be no greater than its corresponding default. Consider including a signed `maxFee` in orders so later administrative changes cannot exceed the maker's authorization.

MX: Fixed in [6c99da](#).

Researcher: Verified fix.

6.3.14 Trading PnL desynchronizes the MarginBank deposit-cap tracker

Context: [MarginBank.sol#L684-L730](#), [MarginBank.sol#L866-L892](#)

Description: `MarginBank` increments `netExternalDeposits[account]` when collateral enters from outside the bank and decrements it whenever that account makes an external withdrawal. Internal transfers used to settle trades change `bankBalances`, but do not change or attribute `netExternalDeposits`.

This makes the cap tracker diverge from the externally deposited principal retained by an account. For example:

1. An account deposits the full 100-token cap, leaving both its bank balance and `netExternalDeposits` at 100.
2. The account earns 100 tokens through a trade, so an operator transfers 100 tokens from a counterparty to it. Its bank balance is now 200 while `netExternalDeposits` remains 100.
3. The account withdraws only the 100-token profit. `_recordExternalWithdrawal` nevertheless clears its tracked external deposits to zero, although the original 100-token deposit remains in the bank.
4. The account can deposit another 100 tokens, ending with a 200-token bank balance while the configured per-account external-deposit cap is 100.

The inverse also occurs: an account that loses or transfers away its original deposit can have a zero bank balance while `netExternalDeposits` remains at the cap, preventing it from replenishing collateral.

If the cap is deliberately intended to measure only lifetime net cash flow (`external deposits - external withdrawals`) regardless of which fungible funds remain, the behavior is internally consistent but should be documented. It does not enforce a cap on retained principal, post-deposit account exposure, or the amount an active trader can replenish.

Recommendation: Define the exact cap invariant and enforce that invariant directly. If the cap is intended to bound account exposure, validate the post-deposit `bankBalances` value. If it is intended to track retained externally funded principal, carry the principal attribution through internal transfers using an explicit cost-basis policy. Otherwise, document that trading gains and losses intentionally decouple the cap tracker from account balances and principal.

MXN: Acknowledged and documented as intended behavior in [bf90413](#).

Researcher: Acknowledged.

6.3.15

Forgiven funding does not attach to the surviving position

Context: [Perpetual.sol#L1624-L1642](#), [Perpetual.sol#L2240-L2314](#), [Perpetual.sol#L2561-L2615](#)

Description: When an account owes funding, settlement consumes its position margin and then any currently positive unrealized PnL. Any remaining debit is recorded as `forgiven`, and the account's local funding index is advanced fully. The unpaid amount is not stored as debt, and the position is neither closed nor frozen.

For example, a one-unit long has entry notional 100, margin 2, and no unrealized PnL at oracle 100. If it owes 10 funding, permissionless settlement consumes 2 and forgives 8. The opposite short can still receive the full 10-token credit, creating an 8-token hole in pooled collateral. The long remains open with zero margin and no record of the missing 8. If price rises to 120 before liquidation or ADL executes, it has 20 tokens of equity that can eventually be closed without repaying the forgiven funding. It can also add new margin after forgiveness and retain the same contingent recovery option.

Because `settleFundingForAccounts()` is permissionless, the debtor can crystallize the write-off at a time when the oracle shows little or no positive PnL, maximizing the amount forgiven.

Reaching the example already implies that funding-settlement and liquidation keepers allowed a debit to grow far beyond posted margin. The zero-margin position is immediately liquidatable/ADL-eligible, and the user cannot execute a trade without a settlement operator, so realizing the recovery option requires a further operational delay or cooperation.

The source comments and `SettlementAmountNotPaidCompletely` event show that writing off uncollectable funding is deliberate. The finding is therefore a design confirmation: it applies only if MXN did not also intend to grant the still-open position all future recovery after that write-off. Monitoring an event does not restore the missing collateral.

Recommendation: Carry unpaid funding forward as account debt and net it against future margin and PnL. If the protocol deliberately cannot represent negative position equity, atomically freeze, liquidate, ADL, or transfer the position before advancing its local index. Do not both forgive the debt and leave the debtor with a live upside option.

MXN: Acknowledged. The unpaid amount is still not attached to the surviving position, so we are not claiming this finding is fixed. The freeze-and-garnish state-machine change remains deferred without a delivery date.

In the meantime, regular operator-triggered account funding settlements realize funding owed instead of waiting for an unrelated trade or margin action. This reduces the time during which a debit can grow before collection, but it does not eliminate the possibility that an account becomes unable to pay between settlements. Forgiven amounts are recorded by `SettlementAmountNotPaidCompletely` and monitored, and the witnessed-bankruptcy gate in [c48d8599](#) prevents a contract-verified bankrupt account from continuing ordinary trading while resolution is pending.

Researcher: Acknowledged.

6.3.16

IsolatedADL inverts the documented taker-side `isBuy` direction

Context: [IsolatedADL.sol#L56-L62](#), [IsolatedADL.sol#L199-L208](#), [IsolatedADL.sol#L311-L357](#), [Types.sol#L70-L98](#), [Perpetual.sol#L106-L123](#)

Description: `IsolatedADL.TradeData.isBuy`, `Types.TradeResult.isBuy`, and `Perpetual's TradeExecuted.isBuy` are all documented as being from the taker's perspective. `_verifyTrade()` instead requires `tradeData.isBuy != makerBalance.isPosPositive`.

The taker and maker must hold opposite positions. If the maker is long, reducing both positions means the maker sells and the short taker buys. The correct taker-side value is therefore `isBuy == true`, which equals the maker's existing direction. The current inequality rejects it. Supplying `false` passes even though it says the taker sold.

Position math ignores this flag and directly reduces both stored quantities, so an operator can work around the revert by always submitting the inverted value. That workaround causes every emitted ADL direction to be opposite the documented taker direction, corrupting event-based trade analytics and any downstream ledger that trusts the event schema.

Recommendation: If the field is intended to remain taker-relative, require `tradeData.isBuy == makerBalance.isPosPositive`. Add long-maker and short-maker tests that assert both state transitions and emitted direction. Alternatively, rename and document the field as maker-relative and invert it when constructing the returned `TradeResult`.

MNX: Fixed in [abfa8bf](#).

Researcher: Verified fix.

6.3.17

Rounding the oracle can make a solvent account ADL-eligible

Context: [IsolatedADL.sol#L128-L180](#), [IsolatedADL.sol#L311-L368](#), [Perpetual.sol#L2638-L2660](#)

Description: `IsolatedADL` rounds the trusted oracle price to the nearest configured tick before checking whether the maker is underwater and the taker is healthy. `Perpetual's` other risk calculations use the raw oracle price. Rounding can therefore change which accounts are eligible for a forced, unsigned ADL.

For example, consider a one-unit long with entry notional 100.06 and margin 0.04. At a raw oracle price of 100.03, the account still has positive equity of 0.01. With a tick size of 0.1, `IsolatedADL` rounds the oracle down to 100.00, reports equity of -0.02, and permits the account to be force-closed at its 100.02 bankruptcy price. The account was solvent at the canonical mark but loses its remaining equity because eligibility used a different price.

The symmetric issue affects shorts when the oracle rounds upward. Near the configured minimum price, rounding can also produce zero; `verifyMinMaxPriceChecks()` then reverts and can disable ADL precisely when a low-priced market needs it.

The direct loss is bounded by approximately half a tick multiplied by quantity, and exploitation requires an authorized settlement operator. If MNX intentionally defines all account health using tick-rounded marks, that rule should be made consistent across `Perpetual` and documented; the current contracts use two different definitions.

Recommendation: Use the unmodified oracle answer for maker/taker health and eligibility. Apply tick rules only to prices that are actually required to be tradable grid prices, and clamp any rounded operational value to valid market bounds.

MNX: Fixed in [abfa8bf](#).

Researcher: Verified fix.

6.3.18

ADL's per-unit epsilon scales into material debt forgiveness

Context: [IsolatedADL.sol#L231-L277](#), [Perpetual.sol#L667-L689](#)

Description: IsolatedADL allows each side's post-trade equity to be negative by as much as $1e14$ raw collateral units per position unit. The comment describes this as a rounding epsilon. Because the check is performed per unit, the permitted shortfall grows linearly with quantity and is not bounded as a total collateral value.

After accepting the negative equity, `_applyIM()` calculates the amount the account would need to pay from free MarginBank collateral. If that `fundsFlow` is positive, the function unconditionally replaces it with zero. The total deficit permitted by the epsilon is therefore silently forgiven rather than collected.

The maximum tolerated deficit is `1e-4 collateral token * total deleveraged quantity`. The PoC uses a one-million-unit low-price position. At the maker's bankruptcy price, the selected short taker has exactly $-1e14$ equity per unit, so the check passes. Across the full quantity, its realized loss exceeds posted margin by 100 whole collateral tokens. `takerFundsFlow` is nevertheless returned as zero, both positions are closed, and the 100-token hole remains socialized in Perpetual's pooled collateral.

An authorized settlement operator must select the harmful taker. Configured maximum and step quantities can force a large position to be processed over multiple legs, but they do not impose a cumulative value cap on the epsilon. The code comments also state that the off-chain planner uses a stricter zero-epsilon check, reducing likelihood under honest operation, but that off-chain convention is not an on-chain bound.

Recommendation: Evaluate solvency from total full-precision equity and require it to be non-negative. If a dust tolerance is operationally necessary, cap the total forgiven value to a small absolute amount independent of quantity, and reject rather than silently discard any material positive funds flow.

MNX: Fixed in [abfa8bf](#).

Researcher: Verified fix.

6.3.19

Replacing the oracle bypasses the configured movement cap

Context: [Perpetual.sol#L1372-L1381](#), [Perpetual.sol#L1863-L1872](#), [Perpetual.sol#L1882-L1924](#), [BackendPriceOracle.sol#L102-L123](#)

Description: Perpetual's ADL replacement-capacity invariant assumes that the oracle cannot move by more than its advertised `maxUpdateBps` between accepted prices. `setOracle()` validates the replacement feed's decimals, positive current answer, future movement cap, and compatibility with the evaluator. It never compares the replacement's current answer with the old oracle's last answer.

Consequently, two feeds that each advertise a 1% per-update cap can be swapped while the old feed reports 100 and the new feed reports 200. The one-time 100% jump passes every setter check. Existing positions are immediately valued at the new answer even though routine updates were constrained to 1%. A feed migration, scale error, or wrong market feed can therefore move maximum-leverage positions past bankruptcy without the transition ever being checked.

The README specifically describes replacement as the path for changing oracle bounds and assigns the setter to the owner, which production plans to make a multisig. This is therefore reportable only if honest migration mistakes are in scope; a malicious trusted owner can already disrupt the market in other ways. Without transition continuity, deploying a replacement also becomes an accidental way to bypass the old and new feeds' bounds for one update.

Recommendation: During `setOracle()`, read both old and replacement answers and require their difference to fit the stricter permitted transition bound, as well as validating freshness and round completeness. If a discontinuous migration is necessary, pause trading, checkpoint funding, and use an explicit governed migration/delist process that accounts for the jump instead of treating it as an ordinary capped update.

MNX: Fixed in [fbc9655](#).

Researcher: Verified fix.

6.3.20 Proportional rounding can make permitted partial liquidation and ADL amounts fail P35

Context: [IsolatedTrader.sol#L643-L692](#), [IsolatedADL.sol#L231-L277](#), [Perpetual.sol#L2638-L2715](#), [BaseMath.sol#L26-L68](#)

Description: A partial position reduction should preserve `oiOpen` and margin in exact proportion to the remaining quantity. Both `IsolatedTrader` and `IsolatedADL` calculate each proportion as `value.baseMul(remainingQuantity).baseDiv(oldQuantity)`. The first operation divides by `1e18` and truncates before the second operation multiplies by `1e18`, losing precision that a single full-precision `value * remainingQuantity / oldQuantity` calculation would retain.

For a distressed account, a mathematically proportional reduction should preserve or improve margin ratio. `Perpetual` nevertheless requires exact `newMarginRatio >= oldMarginRatio` under P35; the liquidation and ADL allowances bypass P37 only, not P35. A small adverse rounding difference therefore reverts the whole forced risk-reduction transaction.

The PoC starts with a long having `qPos = 0.002`, `oiOpen = 0.004`, margin `0.0013333333333333333`, and oracle `1`. Reducing half through `IsolatedADL` produces remaining margin `0.0006666666666666500`. Its margin ratio changes from `-0.333333333333333500` to `-0.3333333333333500000`, which is microscopically worse and fails P35.

This proves that an otherwise permitted partial amount can revert. A full close or a specially chosen quantity may still work. It becomes a hard liveness problem when the position cannot be fully closed in one fill and the configured step/minimum/maximum quantity constraints leave no practical partial amount whose rounding is non-adverse.

Recommendation: Use a single full-precision `mulDiv` for proportional `oiOpen`, margin, and closed-margin calculations. In addition, allow a tightly bounded rounding tolerance in P35 or exempt strictly same-direction quantity reductions that cannot increase exposure.

MNX: Fixed in [f56be0b](#), [abfa8bf](#), [6cff9da](#).

`Perpetual.trade()` now exempts strictly same-direction position reductions from P35's exact margin-ratio comparison, preventing harmless integer-rounding differences from blocking liquidation or automatic-deleveraging reductions. The exemption applies only when every operation for the account in the batch reduces exposure; increases, direction changes, and close-then-add batches remain subject to P35. Separately, `abfa8bf` replaces the double-rounded proportional calculations in `IsolatedTrader` and `IsolatedADL` with one full-precision multiply/divide per field and derives closed values from the exact before/after balances. Regression tests cover the reported rounding vectors and the exemption boundaries.

Researcher: Verified fix.

6.3.21 Vault deposits and withdrawals lack slippage bounds

Context: [LiquidityVault.sol#L419-L457](#), [LiquidityVault.sol#L459-L491](#), [LiquidityVault.sol#L512-L562](#)

Description: `deposit()` accepts only the token amount. It has no `minShares` parameter or deadline, so the transaction succeeds regardless of how few shares are minted after NAV changes. Similarly, a withdrawal request records only the number of shares and its unlock time. It does not record a minimum acceptable token payout, even though the final amount is calculated from NAV much later when `executeWithdrawal()` runs.

For example, suppose the vault has 100 shares and a NAV of 100 tokens. A user submits a 10-token deposit expecting approximately 10 shares. If position value raises NAV to 200 before the transaction executes, the user receives only $10 * 100 / 200 = 5$ shares. The contract provides no way for the user to state that receiving fewer than, for example, 9.9 shares is unacceptable.

For withdrawals, suppose an LP requests redemption of 20 shares when the vault has 100 shares and a NAV of 100 tokens, expecting roughly 20 tokens. If NAV falls to 60 before execution, those shares pay only 12 tokens. Even if the user checks NAV immediately before submitting the execution transaction, an oracle update or market transaction ordered ahead of it in the same block can

reduce the payout, and there is no on-chain minimum that makes the transaction revert.

This remains relevant even if the separate NAV-composition issue is fixed: normal trading, funding, oracle updates, and transaction ordering can still move the share price between quotation and execution.

Recommendation: Add a deposit overload accepting `minShares` and `deadline`, and require both `shares >= minShares` and `block.timestamp <= deadline`. Add `minAssets` to each withdrawal request and require the execution-time payout to meet that stored minimum before burning shares. Alternatively, accept `minAssets` during execution, although storing it in the request more clearly binds the delayed withdrawal to the LP's original authorization. The LP should remain able to cancel an unfillable request without losing shares.

MNX: Acknowledged. This issue is deferred until the LiquidityVault share-pricing redesign is complete. The redesign will ship before LiquidityVault deposits are opened to the public.

Researcher: Acknowledged.

6.3.22

LiquidityVault does not verify the MarginBank collateral token

Context: [LiquidityVault.sol#L179-L203](#), [LiquidityVault.sol#L419-L433](#), [MarginBank.sol#L605-L622](#), [LiquidityVault.sol#L543-L559](#)

Description: `LiquidityVault.initialize()` checks that `_token` is non-zero and has 18 decimals, but it never checks that `_token` is the same token used by `_marginBank`.

During a deposit, LiquidityVault transfers its configured token directly to MarginBank and then asks MarginBank to credit the main vault account. MarginBank does not verify which token was just transferred; it only checks whether its own collateral-token balance is large enough to cover the newly recorded liability.

Normally, a mismatch makes every vault deposit revert because transferring the wrong token does not increase MarginBank's real collateral balance. If MarginBank already has surplus collateral, however, the check can pass and that surplus can be assigned to the vault.

For example, suppose MarginBank uses USDM and has a 100-USDM surplus, while LiquidityVault is mistakenly initialized with an unrelated 18-decimal token. An LP deposits 100 units of the unrelated token. MarginBank receives no new USDM, but its existing surplus allows it to credit the vault with a 100-USDM bank balance. The LP can later redeem and receive 100 real USDM, consuming the bank's surplus in exchange for the wrong token.

Recommendation: During initialization, require `IMarginBank(_marginBank).token() == _token`. Add the `token()` getter to `IMarginBank` if necessary, and perform the check before allowing the vault to be bootstrapped or accept deposits. Also validate that both addresses contain contract code before invoking their interfaces.

MNX: Acknowledged. This issue is deferred until the LiquidityVault share-pricing redesign is complete. The redesign will ship before LiquidityVault deposits are opened to the public.

Researcher: Acknowledged.

6.3.23

A zero-address subaccount approval can make invalid signatures valid

Context: [AccountManager.sol#L65-L67](#), [TypedSignature.sol#L78-L88](#), [Perpetual.sol#L850-L864](#), [Perpetual.sol#L925-L985](#), [Perpetual.sol#L1077-L1090](#)

Description: `AccountManager.setSubAccount()` allows a user to approve `address(0)` as an operator. Separately, `TypedSignature.recover()` uses `ECDSA.tryRecover()` and returns `address(0)` when an ECDSA signature is malformed.

The Perpetual's margin and leverage authorization checks do not first require the recovered signer to be non-zero. They accept the signature when either the recovered signer is the account itself or the signer is an approved subaccount. Together, these behaviors make an invalid signature pass for any account that has approved the zero address.

For example:

1. Alice accidentally, or through a faulty integration, calls `setSubAccount(address(0), true)`.
2. A settlement operator calls `adjustLeverage()` for Alice with an unused nonce and a malformed signature, such as zero-valued `r` and `s`.
3. Signature recovery returns `address(0)`.
4. `isSubAccount(Alice, address(0))` returns `true`, so Alice's collateral is moved without a valid signature.

The same bypass affects `addMargin`, `removeMargin`, and `removeMarginUpTo`. It does not let an arbitrary public caller forge these actions because the entry points are restricted to settlement operators, and it also requires the victim to have approved the zero address. The practical threat is therefore an accidental or integration-created zero-address approval combined with a malicious or compromised settlement operator. The normal order-validation path is not affected because it separately rejects a recovered zero signer.

Recommendation: Reject `address(0)` in `setSubAccount()` and make `isSubAccount()` return `false` for the zero address so any approvals already present in storage cannot be used. As defense in depth, require the recovered signer to be non-zero in every Perpetual signature-authorized function, or make `TypedSignature.recover()` revert whenever `tryRecover()` reports an error. Apply both input and signer validation so a future signature consumer cannot reintroduce the bypass.

MXN: Fixed in [4207f4f](#), [82eca50](#). `AccountManager` now rejects zero-address subaccount approvals and treats any legacy zero-address entries as inactive. All four Perpetual margin/leverage authorization paths independently require a nonzero recovered signer before consulting `AccountManager`, and `MarginBank` applies the same defense to signature-authorized withdrawals. Tests cover malformed signatures, legacy zero entries, and a deliberately permissive replacement `AccountManager`.

Researcher: Verified fix.

6.3.24

An unvalidated trusted forwarder can impersonate users and move their collateral

Context: [MarginBank.sol#L213-L235](#), [BaseRelayRecipient.sol#L31-L48](#), [MarginBank.sol#L484-L522](#)

Description: `MarginBank.initialize()` accepts any address as `_trustedForwarder` and stores it without checking that it is a non-zero contract implementing the expected signature-verifying forwarder logic. There is also no setter, so the address is effectively permanent after initialization.

Calls from this address receive special treatment: `msgSender()` ignores the actual caller identity and treats the last 20 bytes of `calldata` as the user address. A correct forwarder appends an address only after verifying that user's signature. An EOA or malicious contract can append any address without a signature and impersonate that user.

For example, if Bob's EOA is mistakenly configured as the trusted forwarder, Bob can call `requestWithdrawFromBank(Bob, 50e18)` with Alice's address appended to the `calldata`. `MarginBank` records the request as if Alice created it. After the delay, Bob can execute the request with Alice's address appended again, causing 50 tokens to be removed from Alice's bank balance and

sent to Bob. If Alice has left a token allowance, the forwarder can also impersonate her on a deposit call, pull her tokens, and credit an attacker-controlled bank account.

This requires a deployment misconfiguration or compromise of the configured forwarder, but its consequence is complete user impersonation wherever `msgSender()` is trusted.

Recommendation: Accept only a known, audited signature-verifying forwarder deployed by the protocol. At minimum, reject the zero address and addresses with no contract code, while recognizing that a code-length check alone cannot prove the contract is safe. Add an owner-controlled, preferably two-step, forwarder rotation function with an event so a mistaken or compromised forwarder can be replaced or revoked.

MXN: Fixed in [9f64c25](#). The on-chain code gate is in [9f64c25](#), and the previously omitted attestation and guarded rotation path are in [b812aa97](#).

Researcher: Verified fix.

6.3.25 Forwarder can custody and leak native ETH

Context: [MinimalForwarder.sol#L56](#)

Description: The `execute()` function is declared payable and forwards `req.value` to the target, but never checks that the ETH attached to the transaction matches the signed amount, and the contract exposes no receive/withdraw/sweep.

Recommendation: Since no forwarded target needs native value, remove the capability: make `execute` non-payable.

If value forwarding must be retained for future use, instead add `require(msg.value == req.value, 'MinimalForwarder: value mismatch')` at the top of `execute` to prevent overpayment stranding.

MXN: Fixed in [860315a](#). Removed the value field from `ForwardRequest` and made `execute()` non-payable — no forwarded target needs native ETH. The unit test confirms `execute()` rejects attached ETH.

Researcher: Verified fix.

6.3.26 `execute()` increments the nonce before the target call and swallows target reverts

Context: [MinimalForwarder.sol#L64](#)

Description: The `execute()` function in `MinimalForwarder` contract bumps `nonces[req.from]` before invoking the target, and on target revert it returns `(false, returndata)` instead of reverting.

So the nonce increment is committed even though the intended action never happened. A malicious or careless relay can therefore submit a signed request at a moment it is expected to fail (e.g. a `require` in the target won't hold, or `req.gas` is set too low to complete), consuming the user's nonce and invalidating that signature across every relay.

Because the forwarder uses a single sequential nonce, the user must then re-sign at `nonce+1`. The attacker gains nothing and pays the gas, so in isolation this is a Low griefing/liveness issue; it rises toward Medium in time-sensitive situations - e.g. griefing a user's gasless `addMargin` during a volatile price move so their already-signed top-up is dead and must be re-signed and re-submitted, narrowing the window to avoid liquidation.

This is the standard OZ-v4 `MinimalForwarder` trade-off (the return flag is meant to be checked off-chain by an honest relay), which is safe only under an honest-relayer assumption this gasless design does not otherwise make.

Recommendation: Revert on target failure so a failed call rolls back the nonce increment: replace the trailing return (success, returndata) with `require(success)` that bubbles up the target's revert data (e.g. `assembly revert(add(returndata,0x20), mload(returndata)))`, so a request timed to fail cannot consume the nonce and the user keeps their signature. Note this changes the `(bool,bytes)` return contract, so any on-chain caller that reads the success flag must be updated.

MX: Fixed in [860315a](#). `execute()` now bubbles the target's revert instead of returning (`false`, `returndata`), so the nonce increment rolls back with the transaction and a request timed to fail can no longer burn the signer's nonce. The return type changes from (`bool`, `bytes`) to `bytes`. The unit test asserts the nonce is unchanged after a failed target call and that the same signature succeeds once the target recovers.

Researcher: Verified fix.

6.3.27 Non-upgradeable ReentrancyGuard used in upgradeable (UUPS) contracts

Context: [MarginBank.sol#L10](#), [LiquidityVault.sol#L8](#)

Description: Both `LiquidityVault` and `MarginBank` inherit `@openzeppelin/contracts/utils/ReentrancyGuard.sol`, the non-upgradeable guard even though both are UUPS-upgradeable proxy contracts.

Reentrancy protection still functions today (the proxy's `_status` slot defaults to 0, which satisfies the guard's `!= ENTERED` check, so `nonReentrant` works), so this is not a live vulnerability.

However, the non-upgradeable guard contributes a `_status` storage slot with no reserved gap, baking that slot into the contracts' storage layout. Any future migration to `ReentrancyGuardUpgradeable`, removal of the guard, or reordering of the inheritance chain would shift every subsequent state variable and silently corrupt storage on upgrade. This is a recognized anti-pattern that OpenZeppelin's Upgrades tooling flags for upgradeable contracts.

Recommendation: Replace `ReentrancyGuard` with `ReentrancyGuardUpgradeable` in both contracts and call its `__ReentrancyGuard_init()` from each contract's initialize function, ensuring the guard participates correctly in the upgradeable storage-gap scheme.

MX: Fixed in [0f1bfb3](#), [6cfd34b](#). On our pinned OpenZeppelin 5.6.1, `ReentrancyGuardUpgradeable` and `__ReentrancyGuard_init()` do not exist, while `ReentrancyGuard` is stateless in the sequential layout and stores its state at a fixed ERC-7201 namespaced slot. It therefore cannot shift subsequent storage when inherited, reordered, or removed, and the proxy's zero-valued default is a valid non-entered state. Added `ReentrancyGuardProxyCompatibility.test.ts` to pin the OpenZeppelin version, assert the guard's empty storage layout, anchor layouts across every guard-bearing proxied contract, and test `MarginBank` and `LiquidityVault` behind real proxies. We'll evaluate `ReentrancyGuardTransient` with OpenZeppelin v6 once EIP-1153 support is confirmed on our target chain.

Researcher: Verified fix.

6.3.28 Unvalidated, immutable guardianContract can permanently disable the withdrawal-halt safety valve

Context: [MarginBank.sol#L228](#)

Description: `MarginBank.initialize` assigns `guardianContract = _guardian` with no zero-address validation, in contrast to the `_signatureVerifier` parameter which is explicitly checked (`require(_signatureVerifier != address(0), ...)`). `guardianContract` is written only at initialization and there is no setter anywhere in the contract. So its value is effectively immutable after deployment.

The guardian's sole authority is `setWithdrawalStatus`, gated by `onlyGuardianContract` modifier, which is the emergency switch that halts all withdrawals via `isWithdrawalAllowed()`; this gate covers direct, signature-authorized, and system-account withdrawals, including LP redemptions.

If the contract is deployed with `_guardian` set to `address(0)` (or any incorrect address), `setWithdrawalStatus` becomes permanently unreachable or misassigned, and the protocol loses its ability to freeze withdrawals during an incident with no remedy short of a full contract upgrade.

Recommendation: Validate `require(_guardian != address(0), 'MarginBank: invalid guardian')` in `initialize`, and add an owner-gated `setGuardianContract(address)` setter (with a non-zero check and an emitted event) so the guardian can be corrected or rotated without redeploying/upgrading.

MX: Fixed in [33bd129](#). `initialize` now requires a non-zero guardian, and a new owner-gated `setGuardianContract` (zero-check + `GuardianContractUpdate` event) lets the withdrawal-halt authority be corrected or rotated without an upgrade.

Researcher: Verified fix.

6.3.29 `depositFullAllowanceToBank` reverts on infinite (`type(uint256).max`) approvals

Context: [MarginBank.sol#L398-L401](#)

Description: `depositFullAllowanceToBank` reads the caller's allowance and rejects it via `require(approvedAmount <= type(uint128).max, 'MarginBank: allowance too large')`. Because many users (and wallets) set a one-time `type(uint256).max` "infinite" approval, this check reverts for the most common approval pattern, making the function unusable for such callers even though their actual deposit would be far below `uint128` range.

The function is documented for routed exact-approval flows, so its intended caller is unaffected, but a regular user with a standing infinite approval who calls it hits a confusing revert rather than a deposit.

Simply dropping the cap would not help truncating `type(uint256).max` to `uint128` yields `type(uint128).max`, and the subsequent `transferFrom` of that amount would revert on insufficient balance.

Recommendation: When the allowance is the infinite sentinel (`type(uint256).max`), `deposit` `min(tokenBalanceOf(caller), type(uint128).max)` instead of reverting, so infinite-approval users can deposit their full balance; guard this to the exact sentinel and document that it sweeps the caller's entire balance. Alternatively, keep the revert but document that infinite-approval callers must use `depositApprovedToBank/depositToBank` with an explicit amount.

MX: Kept by design; documented in [33bd129](#). `depositFullAllowanceToBank` is intended only for routed exact-allowance flows (the Daimo Pay router); auto-sweeping an infinite approval would be a worse failure mode and conflicts with the per-account external deposit cap. The revert message now names the exact-amount alternative (`depositToBank`) and `NatSpec` documents the intended caller.

Researcher: Acknowledged.

6.3.30 `setDirectWithdrawalDelaySeconds` has no upper bound, and its lower floor only catches the 0 sentinel

Context: [MarginBank.sol#L274](#)

Description: `setDirectWithdrawalDelaySeconds` assigns `directWithdrawalDelaySeconds`, the delay between requesting and executing a direct bank withdrawal with no bounds.

On the high end, an excessively large value (up to `type(uint64).max`) pushes `executableAt` so far out that direct withdrawals become de-facto permanently locked, trapping user funds; this is reachable via a compromised owner or a fat-finger.

On the low end, the request path floors the delay to 60 seconds only when the stored value is exactly 0 (if `delaySeconds == 0`) `delaySeconds = 60`), so an owner can still set any small non-zero value (e.g. 1) to obtain a near-instant delay that undershoots the contract's own implied 60-second minimum, weakening the reaction/dispute window the delay exists to provide.

The zero-sentinel floor is therefore inconsistent, it protects the uninitialized case but not explicitly-set sub-minimum values.

Recommendation: Enforce sane bounds in the setter, e.g. `require(_delaySeconds >= MIN_WITHDRAWAL_DELAY && _delaySeconds <= MAX_WITHDRAWAL_DELAY, 'MarginBank: delay out of range')`, with a non-zero minimum that preserves the

protective window and a maximum that cannot indefinitely trap user withdrawals.

MNX: Fixed in [33bd129](#). `setDirectWithdrawalDelaySeconds` now enforces $60s \leq \text{delay} \leq 7 \text{ days}$. The request-time $== 0 \rightarrow 60s$ floor is retained as a backstop for any legacy stored value.

Researcher: Verified fix.

6.3.31 Missing `_operator` validation and unimplemented contract-type check in `setBankOperator`

Context: [MarginBank.sol#L250](#)

Description: `setBankOperator` grants the bank-operator role - which can move margin between arbitrary accounts via `transferMarginToAccount` / `transferMarginToAccountSilent` but performs no validation on `_operator` and still carries an unimplemented // TODO implement EIP165 to ensure operator is PERPETUAL or LIQUIDATION contract.

This is inconsistent with every sibling privileged setter (`setSystemAccount`, `setSystemAccountController`, `setSignatureVerifier`), all of which reject `address(0)`.

As written, an owner typo or wrong address can silently approve `address(0)` or an EOA / arbitrary contract as a fund-moving operator; because operators can relocate any account's margin, a misconfiguration here has a large blast radius despite being owner-gated.

Recommendation: Add `require(_operator != address(0), 'MarginBank: invalid operator')`, and resolve the TODO by verifying via ERC-165 `supportsInterface` (or an equivalent registry/interface check) that `_operator` implements the expected Perpetual/Liquidation operator interface before approval.

MNX: Fixed in [33bd129](#). `setBankOperator` now rejects the zero address and requires the operator to have contract code on approval — an EOA operator is never legitimate since operators can move any account's margin. The stale EIP-165 TODO is replaced with a rationale comment: our operator contracts don't implement ERC-165, so the code-existence check plus owner gate plus event is the intended control.

Researcher: Verified fix.

6.3.32 Unbounded, never-shrinking `activeChildMarkets` array makes `totalNAV()` $O(N^2)$ and DoS prone

Context: [LiquidityVault.sol#L645-L667](#)

Description: Each child's markets are tracked in an `address[] activeChildMarkets[child]` array whose entries are appended on activation but never removed on deactivation. `setChildActiveForMarket(child, market, false)` only flips the `isActiveChildMarket` bool, leaving the array to grow monotonically.

On top of that, `totalNAV()` iterates this array and, for every element, runs `_isFirstTrackedMarketOccurrence`, an inner $O(N)$ scan, yielding $O(N^2)$ work per child.

Because `totalNAV()` is on the critical path of both `deposit()` (called twice) and `executeWithdrawal()`, a child that has been activated/deactivated across many markets over time can inflate the loop until it exceeds the block gas limit, permanently bricking share pricing and thus deposits and withdrawals. This compounds the lack of an upper bound on child-vault count.

Recommendation: Replace `activeChildMarkets` (and the parallel `isActiveChildMarket` mapping) with `EnumerableSet.AddressSet`, giving $O(1)$ add/remove/contains and removing entries on deactivation; this collapses the NAV loop to $O(\text{active markets})$ and eliminates `_isFirstTrackedMarketOccurrence`.

MNX: Fixed in [ba51f97](#). Market deactivation now swap-removes the entry, so each child's array contains only active markets. A storage-compatible reinitializer compacts historical arrays after upgrade. The contract stores the global active-pair counter; the per-child count is the compacted array length exposed by `activeChildMarketCount()`. The resulting traversal is linear in currently

active child/market pairs and bounded by the 512-per-child and 1,024-global limits. Upgrade and boundary tests cover legacy duplicates, inactive entries, counters, and cap enforcement.

Researcher: Verified fix.

6.3.33

Setting a deposit cap below current net deposits blocks all new deposits

Context: [LiquidityVault.sol#L262](#), [LiquidityVault.sol#L267](#)

Description: `setTotalVaultDepositCap` (and likewise `setPerDepositorVaultDepositCap`) performs no validation that the new cap is greater than or equal to the current `totalNetVaultDeposits`.

Since `_recordVaultDeposit` enforces `nextTotalDepositAmount = totalNetVaultDeposits + amountToken <= totalVaultDepositCap`, any non-zero cap set below the existing net deposits ($0 < \text{cap} < \text{totalNetVaultDeposits}$) causes every subsequent `deposit()` to revert with 'total cap exceeded', regardless of amount.

This effectively freezes all new deposits until enough LP withdrawals reduce `totalNetVaultDeposits` back below the cap. The condition is applied silently the setter neither reverts nor emits any distinct signal, so operators and off-chain monitoring only discover the freeze through failing user transactions.

Recommendation: Validate that a non-zero cap is $\geq \text{totalNetVaultDeposits}$ in `setTotalVaultDepositCap` (and $\geq$ the relevant per-depositor total in `setPerDepositorVaultDepositCap`), or explicitly document this freeze behavior and surface it via a dedicated event/flag rather than relying on deposit-time reverts.

MNX: Fixed in [ba51f97](#). Both deposit-cap setters may intentionally lower a cap below existing tracked deposits. This blocks subsequent deposits until withdrawals restore sufficient headroom, while withdrawals remain available. NatSpec documents this behavior and clearly states that zero disables the cap rather than freezing deposits; the incident pause switches are the freeze mechanism. Regression tests cover both the per-depositor and global caps: lowering below existing deposits, blocked deposits, successful withdrawals, reopened headroom, and zero restoring uncapped deposits.

Researcher: Verified fix.

6.3.34

Missing upper bound validation on `maxChildVaults`

Context: [LiquidityVault.sol#L186](#), [LiquidityVault.sol#L233](#)

Description: Neither `initialize` nor `setMaxChildVaults` enforces an upper bound on `maxChildVaults` (the latter only checks `value >= childVaults.length`).

`totalNAV()` iterates over every registered child vault and, for each, over its active markets, and is invoked on every `deposit()` (twice) and `executeWithdrawal()`.

If `maxChildVaults` is set to a large value and many children are created, the unbounded loop can exceed the block gas limit, permanently bricking share pricing and thus deposits and withdrawals. Because share accounting depends on `totalNAV()`, this becomes a griefing/DoS vector rather than a recoverable misconfiguration.

Recommendation: Consider introducing an upper bound validation.

MNX: Fixed in [ba51f97](#). We now enforce hard limits of 64 child vaults, 512 active markets per child, and 1,024 active child/market pairs globally. These limits apply during initialization, configuration changes, market activation, and the legacy-state migration. Boundary and upgrade tests cover their enforcement.

Researcher: Verified fix.

6.3.35 Missing validation of `withdrawLockMs` in `initialize`

Context: [LiquidityVault.sol#L185](#)

Description: The `initialize` function validates all address parameters but never validates `_withdrawLockMs` before assigning it to the `withdrawLockMs` state variable.

If set to zero, the withdrawal timelock is effectively disabled: `requestWithdrawal()` computes `unlockAtMs = block.timestamp * 1000 + 0`, so `executeWithdrawal()` can be called in the same block.

This removes the delay that lets the vault react to NAV changes or incidents (e.g. enabling `withdrawalsPaused`) before LPs exit, and can let withdrawals front-run adverse NAV updates. Because the vault is upgradeable and initialized once, an erroneous zero value persists until a manual fix.

Recommendation: Require `_withdrawLockMs` to be within a sane, non-zero range (e.g. `require(_withdrawLockMs >= MIN_LOCK && _withdrawLockMs <= MAX_LOCK)`) in `initialize`, and consider adding a bounded owner setter for future adjustments.

MNX: Acknowledged. This issue is deferred until the LiquidityVault share-pricing redesign is complete. The redesign will ship before LiquidityVault deposits are opened to the public.

Researcher: Acknowledged.

6.4 Informational

6.4.1 Dead declarations obscure the Phase 3 control surface

Context: [Guardian.sol#L68-L75](#), [Guardian.sol#L8-L19](#), [FundingOracle.sol#L8-L17](#), [Evaluator.sol#L9-L29](#)

Description: Guardian declares `POStatusUpdate`, described as reporting when price-oracle updates are enabled or disabled. The contract has no corresponding price-oracle status storage, setter, or emission site. Its implemented controls cover trading, funding, and withdrawals only. The dead event makes the public ABI appear to expose an emergency oracle control that does not exist.

The reviewed contracts also retain compile-time declarations with no runtime use:

- Guardian imports `BaseMath` but never references it.
- FundingOracle imports `IGuardian` but never references it.
- Evaluator imports `PositionBalanceMath` and installs a `using PositionBalanceMath` for `Types.PositionBalance` directive, but invokes none of the library's methods.

These items do not directly affect protocol behavior, but they obscure the actual dependency and emergency-control surface and add noise to static-analysis and maintenance workflows.

Recommendation: Remove the dead event unless oracle pausing is an intended feature that will be implemented and tested. Remove unused imports and the unused `using` directive. For deployed ABI compatibility, document a retained event as deprecated and never emitted rather than leaving integrators to infer a missing control.

MNX: Fixed in [d90da0b](#).

Researcher: Verified fix.

6.4.2

FundingRateUpdate mislabels a remaining duration as an absolute timestamp

Context: [FundingOracle.sol#L58-L64](#), [FundingOracle.sol#L243-L259](#), [FundingOracle.sol#L316-L324](#)

Description: The `FundingRateUpdate` event calls its third field `minApplicationTime` and documents it as the minimum times-tamp through which the rate applies. Both emission sites instead populate it with:

```
nextWindowBoundary - currentTimestamp
```

The value is a remaining duration, normally between zero and 3,600 seconds, rather than a Unix timestamp. The fourth field separately contains the actual current timestamp.

An indexer or funding scheduler following the ABI documentation can interpret a value such as 1800 as an absolute time in 1970 and conclude that the rate has already expired. Conversely, a consumer that discovered the duration behavior empirically will break if the implementation is later changed to match the existing comment. This does not change on-chain funding accounting, but it makes event-driven monitoring and independent funding reconciliation ambiguous.

Recommendation: Decide on one semantic and encode it unambiguously. Either rename the field to `remainingApplicationSeconds`, or emit the absolute next-window timestamp promised by the current name and documentation. Include the unit in `NatSpec` and add an event assertion covering a midpoint within a funding window.

MNX: Fixed in [e4a6150](#).

Researcher: Verified fix.

6.4.3

IEvaluator omits view mutability from read-only methods

Context: [IEvaluator.sol#L6-L75](#), [IEvaluator.sol#L77-L125](#), [Evaluator.sol#L224-L243](#)

Description: `Evaluator` implements all validation routines as `view`, but `IEvaluator` declares them without `view`. The interface also omits `view` from the `tickSize`, `minQty`, `maxQtyLimit`, `maxQtyMarket`, `stepSize`, and `maxAllowedOpenNotional` getters even though they are generated from read-only public storage.

Calls compiled against the interface are consequently ordinary `CALLs` rather than `STATICCALLs`. The current `Evaluator` implementation does not mutate state, so this creates no direct exploit in the reviewed deployment. It does, however, remove EVM-enforced read-only behavior at the consumer boundary, causes ABI tooling to classify getters and validators as state-changing, and forces helpers such as `IsolatedADL._roundPrice()` to remain non-view merely because `tickSize()` is misdeclared. A replacement or mock `Evaluator` could also perform side effects during a validation call that consumers reasonably treat as read-only.

Recommendation: Add `view` to every validation method and storage getter in `IEvaluator` whose implementation is read-only. Regenerate downstream ABIs and mark helper functions such as `_roundPrice()` as `view` after the interface is corrected.

MNX: Fixed in [6cff9da](#).

Researcher: Verified fix.

6.4.4

Dead state and events obscure the trader interfaces

Context: [IsolatedTrader.sol#L227-L244](#), [IsolatedTrader.sol#L306-L326](#), [IsolatedADL.sol#L43-L51](#), [IsolatedADL.sol#L78-L87](#), [IsolatedADL.sol#L102-L117](#), [Perpetual.sol#L93-L98](#)

Description: The scoped settlement contracts retain several public state variables and events that have no runtime role:

- `IsolatedTrader.marginBank` is assigned from an initializer argument but is never read.
- `IsolatedADL.marginBank` is declared as a public address but is neither initialized nor read, so its getter permanently reports the zero address.
- `IsolatedADL.OrderCancel` is declared even though ADL has no order-cancellation flow and never emits it.
- `Perpetual.SubAccountUpdate` is declared but never emitted; subaccount authorization is handled through `AccountManager`.

These declarations do not create a direct security impact, but they enlarge the ABI and storage surface, suggest integrations or observability that do not exist, and make deployment validation ambiguous. In particular, consumers may incorrectly treat the public `marginBank` getters as authoritative dependencies.

Recommendation: Remove unused declarations where storage compatibility permits. For already deployed upgradeable storage, retain the slot but rename or document it as deprecated rather than reusing it. Remove the unused initializer argument and dead events from new deployments, and ensure interfaces expose only live behavior.

MNX: Fixed in [96cdcb31](#).

Researcher: Verified fix.

6.4.5

BaseMath narrowing casts silently wrap results outside the assumed economic domain

Context: [BaseMath.sol#L28-L33](#), [BaseMath.sol#L61-L85](#), [BaseMath.sol#L102-L123](#), [BaseMath.sol#L139-L143](#)

Description: Several shared `BaseMath` helpers calculate in `uint256` or `int256` and then explicitly narrow the result to `uint128`. Solidity's explicit narrowing conversions do not revert when the value is too large; they retain only the low 128 bits.

For example, `baseMul(type(uint128).max, type(uint128).max)` computes an intermediate value that does not fit in `uint128`, then returns a truncated value instead of signalling that the fixed-point result is outside its supported domain. `baseDiv`, the round-up variants, `absolute`, and `toPositive` have the same implicit result-size assumption. `absolute(type(int256).min)` additionally reverts while attempting to negate the minimum signed integer.

Current prices, quantities, margins, and capped default fee rates are expected to remain far below these boundaries, so no ordinary unprivileged path was identified that reaches the casts with overflowing values. Some configurable inputs, including custom trader fees, are not bounded by those default limits and weaken this assumption. As shared primitives, the helpers expose no explicit contract documenting or enforcing their valid input and result range.

Recommendation: Use a checked cast such as `OpenZeppelin SafeCast.toUint128` after computing each wide result, and handle `type(int256).min` explicitly in `absolute`. Document the maximum supported fixed-point domain and enforce economically meaningful bounds at every external configuration boundary.

MNX: Fixed in [3d16462](#).

Researcher: Verified fix.

6.4.6

Privileged configuration changes omit events

Context: [LiquidityVault.sol#L233-L236](#), [Perpetual.sol#L1475-L1483](#)

Description: `LiquidityVault.setMaxChildVaults` and `Perpetual.setAccountManager` mutate privileged configuration without emitting events.

Changing `maxChildVaults` alters how many custody-bearing system accounts the vault can deploy. Changing `accountManager` replaces the dependency used for account and subaccount authorization. Both changes are relevant to operational monitoring and incident response, but neither can be detected from logs; indexers must continuously poll storage or infer the change from later behavior.

The omission is inconsistent with nearby setters, which emit dedicated events for guardian, whitelist, fee, settlement-operator, and trade-contract changes.

Recommendation: Add events containing both previous and new values, such as `MaxChildVaultsUpdate(previousValue, newValue)` and `AccountManagerUpdate(previousAccountManager, newAccountManager)`, and emit them after successful mutation.

MNX: Fixed in [ba51f97](#) and [dfc265c](#). `LiquidityVault.setMaxChildVaults` emits `MaxChildVaultsUpdate(previousValue, newValue)`, and `Perpetual.setAccountManager` emits the indexed `AccountManagerUpdate(previousAccountManager, newAccountManager)` declared through the shared interface. Tests assert exact previous/new values together with the existing authorization and validation behavior.

Researcher: Verified fix.

6.4.7

LiquidityVault initialization whitelists the caller instead of the owner

Context: [LiquidityVault.sol#L179-L215](#)

Description: `LiquidityVault.initialize` assigns ownership to the explicit `_owner` argument and enables the deposit whitelist, but the single initially whitelisted depositor is `_msgSender()` rather than `_owner`.

When a factory, deployment script contract, or proxy administrator initializes the vault for a different operational owner, the factory is granted deposit permission while the designated owner is not. The owner must submit a separate configuration transaction before it can bootstrap liquidity through the ordinary deposit flow. If the deployer retains funds or approvals, its unintended permission also remains active until explicitly revoked.

The owner can repair the whitelist immediately, so this is a deployment footgun and least-privilege inconsistency rather than a permanent denial of service.

Recommendation: Whitelist `_owner` during initialization, or accept an explicit `initialDepositor` argument and validate it as non-zero. If the initializer caller is intentionally supposed to fund the vault, document that requirement and revoke factory permissions as part of the atomic deployment workflow.

MNX: Fixed in [64052cd](#).

Researcher: Verified fix.

6.4.8

Maker order cancellation cannot use the trusted forwarder

Context: [BaseRelayRecipient.sol#L6-L9](#), [BaseRelayRecipient.sol#L35-L52](#), [IsolatedTrader.sol#L475-L486](#), [IsolatedTrader.sol#L507-L515](#)

Description: `IsolatedTrader` inherits `BaseRelayRecipient`, whose integration requirement is to use `msgSender()` wherever the effective caller is needed. `cancelOrder` instead compares and logs `OpenZeppelin Context's _msgSender()`.

For a call routed through the trusted forwarder, `_msgSender()` is the forwarder contract rather than the maker appended to the calldata. The maker check therefore reverts even when the forwarded request was correctly authorized. `cancelOrders` inherits the same behavior because it calls `cancelOrder` for every supplied order.

Direct cancellations still work, and the separate cancellation-operator path correctly uses a relay-aware modifier. The impact is limited to the advertised meta-transaction surface: makers cannot have their own cancellation transaction relayed and must hold native gas or rely on the privileged cancellation operator.

Recommendation: Replace `_msgSender()` with `msgSender()` in both the authorization check and `OrderCancel` event. Add a regression test that submits a maker cancellation through `MNXMinimalForwarder` and verifies that the maker, rather than the forwarder, is recorded.

MNX: Fixed in [6cff9da](#).

Both `cancelOrder` and `cancelOrders` now authorize and emit the relay-aware `msgSender()` maker rather than the raw forwarder caller. A correctly signed forwarded cancellation therefore behaves exactly like a direct maker cancellation, while an appended wrong maker and unauthorized caller still fail.

Researcher: Verified fix.

6.4.9

BatchTradeExecuted reports the first fill rather than the documented net direction

Context: [Perpetual.sol#L125-L136](#), [Perpetual.sol#L799-L814](#), [Perpetual.sol#L2329-L2352](#)

Description: The `BatchTradeExecuted.isLong` field and its `TraderAccumulator` counterpart are documented as the user's net position direction for the batch. `_accumulateTradeResult` instead writes `isLong` only for the user's first fill and leaves it unchanged for all later fills.

If a user buys in one match and then sells or flips direction in a later match in the same batch, the event continues to report the direction of the first match. It can therefore report `isLong = true` even when the user's net batch flow or final position is short, or report a direction for a batch whose buys and sells net to zero.

The on-chain position is updated independently, so this does not corrupt settlement. It does make the event inconsistent with its documented semantics and can mislead indexers, accounting pipelines, risk monitoring, or user interfaces that rely on the aggregate event instead of reconstructing every match.

Recommendation: Define whether the field represents first-fill direction, net traded direction, or final position direction. Compute and emit the selected value from the aggregate signed quantity or final running position. If retaining the existing behavior, rename the field to `firstFillIsLong` and update its `NatSpec` so consumers do not interpret it as a net value.

MNX: Fixed in [6cff9da](#). We retained the existing first-fill behavior and corrected the public contract to match it: the event and accumulator field are now named `firstFillsIsLong`, with explicit documentation that later fills do not revise it. Parameter names are not part of an event's byte-level signature, so existing log topics and encoded data remain compatible while generated a

Researcher: Verified fix.

6.4.10 Redundant `depositApprovedToBank` duplicates `depositToBank`

Context: [MarginBank.sol#L414](#)

Description: `depositApprovedToBank(address _account, uint128 _amount)` has a body identical to `depositToBank(address _account, uint128 _amount)`: both resolve `caller = msgSender()`, call `_depositFrom(caller, _account, _amount)`, and return `true`.

The two share the same signature and behavior, differing only in function name (and therefore selector) and NatSpec intent. `depositApprovedToBank` is documented for "exact-amount contract-call flows," which is precisely what `depositToBank` already performs.

The distinction is nominal, providing no behavioral difference while enlarging the external interface, adding a second selector to audit and maintain, and inviting integrator confusion over which function to call.

Recommendation: Remove `depositApprovedToBank` and its `IMarginBank` interface entry in favor of `depositToBank`; if it must be retained for API ergonomics or backward compatibility, document it explicitly as an exact alias of `depositToBank`.

MX: Fixed in [33bd129](#). Removed `depositApprovedToBank` and its `IMarginBank` entry; it duplicated `depositToBank`, which callers should use.

Researcher: Verified fix.

6.4.11 `depositToBankWithPermit` supports relaying only via ERC-2771, not standalone permit-relay

Context: [MarginBank.sol#L363](#)

Description: `depositToBankWithPermit` binds the EIP-2612 permit owner to `msgSender()` (`permit(caller, address(this), ...)` where `caller = msgSender()`), so the permit signature must belong to whoever the contract resolves as the sender.

A relayer that forwards only the user's permit signature by calling the function directly (not through the trusted forwarder) resolves `msgSender()` to the relayer, so permit validates the user's signature against the wrong owner and reverts.

Gasless relaying therefore works exclusively through the ERC-2771 trusted-forwarder path, where `msgSender()` returns the original signer extracted from calldata.

This is a deliberate safety constraint rather than a defect: the ERC-2771 meta-transaction signature authenticates the entire calldata including the deposit destination `_account` whereas a design taking an explicit owner parameter would let a relayer pull the user's permit-authorized tokens and credit them to an `_account` the user never chose (the permit authorizes only the amount, not the destination).

Recommendation: Document that gasless deposits must be relayed through the ERC-2771 forwarder path, so integrators do not attempt standalone permit-relaying (submitting only the user's permit signature via a direct call) and encounter reverts; optionally note the destination-binding rationale so the constraint is understood as intentional.

MX: Acknowledged as intended; documented in [33bd129](#). NatSpec now states that gasless permit deposits must be relayed through the ERC-2771 forwarder, and why the permit owner is bound to `msgSender()`: the ERC-2771 signature authenticates the full calldata including the destination account, whereas an explicit-owner parameter would let a relayer redirect permit-authorized funds to an account the user never chose.

Researcher: Acknowledged.

6.4.12

Guardian-controlled `setWithdrawalStatus` emits no event

Context: [MarginBank.sol#L265](#)

Description: `setWithdrawalStatus` mutates `'withdrawalStatus`, the guardian-controlled flag that gates whether funds can be withdrawn from the `MarginBank` (via `isWithdrawalAllowed()`) but emits no event, unlike every other privileged setter in the contract, all of which emit on state change.

Because halting or resuming withdrawals is a high-signal incident-response action, this omission leaves off-chain monitoring, dashboards, and users unable to detect from logs when the withdrawal gate is toggled or by whom; they can only observe it by polling storage / `isWithdrawalAllowed()`.

It also breaks the contract's own consistent emit-on-mutation convention, making the change harder to audit and correlate during an incident.

Recommendation: Define and emit a dedicated event (e.g. `WithdrawalStatusUpdate(Types.GuardianStatus previousStatus)`) inside `setWithdrawalStatus` so guardian-driven withdrawal halts and resumptions are observable on-chain.

MNX: Fixed in [33bd129](#). `setWithdrawalStatus` now emits `WithdrawalStatusUpdate(previousStatus, newStatus)`, so guardian-driven halts and resumptions are observable on-chain.

Researcher: Verified fix.

6.4.13

Redundant `isolatedMargin` parameter and tautological margin check in `inheritPosition`

Context: [LiquidityVault.sol#L594](#)

Description: `inheritPosition` accepts a standalone `isolatedMargin` argument and enforces `require(isolatedMargin == position.margin, 'LiquidityVault: margin mismatch')`, but the only on-chain caller: `Perpetual.sol:1994` passes `position.margin` itself as that argument.

The check is therefore a tautology that can never fail in practice, and the parameter is redundant since `position.margin` (documented in `Types.PositionBalance` as the margin/collateral locked in the position) is already available.

The value is additionally emitted twice in `BackstopInheritanceAuthorized` once inside the position struct and again as `isolatedMargin` duplicating log data. The net effect is wasted calldata, an extra interface parameter, and a consistency guard that provides no real protection given the single existing caller.

Recommendation: Remove `isolatedMargin` from `IBackstopVault.inheritPosition` and `LiquidityVault`, using `position.margin` directly. If the interface must accommodate future markets that track isolated margin separately from `position.margin`, document that intent explicitly and retain the cross-check as a deliberate invariant rather than incidental redundancy.

MNX: Fixed in [ba51f97](#). The redundant argument was removed from `IBackstopVault`, `LiquidityVault`, the event, and the `Perpetual` caller. A legacy-proxy upgrade test validates storage compatibility and the new selector behavior. Because this changes a cross-contract selector, deployment will upgrade the vault and all registered perpetual markets as one paused campaign.

Researcher: Verified fix.

6.4.14

Path-dependent ceil rounding in `_recordVaultWithdrawal` lets split withdrawals free extra deposit-cap

Context: [LiquidityVault.sol#L799](#)

Description: `_recordVaultWithdrawal` reduces a depositor's `netVaultDeposits` by `Math.mulDiv(currentDeposits, sharesBurned, sharesBefore, Ceil)`, rounding the reduction up.

Because each partial withdrawal rounds up independently, the total cap freed is path-dependent: burning the same shares across several smaller withdrawals frees strictly more principal than burning them in one call (fuzz-confirmed - e.g. burning 2 of 10 shares against a principal of 5 frees 1 in one shot but 2 when split into two 1-share burns).

Since `netVaultDeposits` gates the per-depositor and total deposit caps, this lets a depositor recover marginally more cap headroom than the proportional amount by fragmenting withdrawals, weakening cap enforcement.

The effect is bounded to at most 1 wei per withdrawal and every withdrawal is timelocked and pays out real value, so exploiting it to any meaningful degree is impractical; funds and share/payout accounting are unaffected.

Recommendation: If strict cap enforcement is desired, round the reduction down (`Math.Rounding.Floor`) so withdrawals never free more than the proportional principal; any sub-wei dust left behind is cleared by the existing full-exit branch (`sharesBurned >= sharesBefore`). Otherwise, document the Ceil choice and its path-dependence as intended (dust-avoidance) behavior.

MNX: Fixed in [ba51f97](#). Partial withdrawals now reduce tracked principal using floor rounding, preventing repeated partial withdrawals from freeing more cap headroom than the withdrawn proportion warrants. The existing full-exit path continues to clear tracked principal completely. Tests cover partial rounding and full exits to zero.

Researcher: Verified fix.

6.4.15

Redundant intermediate variable `payoutE18` in `executeWithdrawal`

Context: [LiquidityVault.sol#L535-L536](#)

Description: In `executeWithdrawal`, `payoutE18` is computed as `(shares * navBefore) / totalShares` and then immediately copied into `tokenAmount` (`uint256 tokenAmount = payoutE18;`), after which `payoutE18` is never referenced again. T

he two variables are always numerically identical because the vault hard-requires 18-decimal collateral, so the intended "internal E18 unit vs token unit" distinction collapses to an identity conversion. The extra local adds no behavior or safety and only obscures that the values are the same.

Recommendation: Assign directly to `tokenAmount` (`uint256 tokenAmount = (shares * navBefore) / totalShares;`) and drop `payoutE18`, or, if the unit seam is intentionally kept for future non-18-decimal support, document it with a comment.

MNX: Acknowledged. This issue is deferred until the LiquidityVault share-pricing redesign is complete. The redesign will ship before LiquidityVault deposits are opened to the public.

Researcher: Acknowledged.

6.4.16 Redundant `totalNAV()` recomputation in `deposit()` for an event-only, derivable value

Context: [LiquidityVault.sol#L445](#)

Description: `deposit()` invokes `totalNAV()` a second time to populate the `navAfter` field of the `VaultDeposited` event, even though `navAfter` is exactly `navBefore + amountToken`, since the only NAV-affecting state change between the two snapshots is the main vault balance increasing by `amountToken` via `depositStrandedSystemAccountToken`, with no child balance, market equity, or trade activity in between.

Both `navBefore` and `amountToken` are already emitted in the same event, so `navAfter` is fully reconstructable off-chain.

Because `totalNAV()` is an $O(\text{children} \times \text{markets})$ traversal (with an additional $O(N^2)$ dedup scan) that performs an external balance/equity read per entry, this second call roughly doubles the deposit's NAV-read gas and compounds the existing NAV-loop gas/DoS surface, all for a value that carries no on-chain benefit and is guarded by no invariant check.

Recommendation: Replace the second traversal with `uint256 navAfter = navBefore + amountToken`; (or drop the `navAfter` field entirely, since it is derivable from the emitted `navBefore` and `amountToken`).

MNX: Acknowledged. This issue is deferred until the LiquidityVault share-pricing redesign is complete. The redesign will ship before LiquidityVault deposits are opened to the public.

Researcher: Acknowledged.

6.4.17 ChildVaultCreated omits the nonce used to derive the child address

Context: [LiquidityVault.sol#L726](#)

Description: Child vault accounts are deployed via `CREATE2` with a salt of `keccak256(chainid, address(this), label, childVaultNonce)`, and since labels are not enforced to be unique, the `childVaultNonce` is the only factor that distinguishes two children sharing the same label.

The `ChildVaultCreated(child, label, trader)` event emits the resulting address and label but not this nonce. While the child address is itself the unique on-chain identifier, omitting the nonce prevents off-chain consumers from independently reconstructing/verifying the `CREATE2` salt and address derivation, and from cleanly ordering or correlating same-labelled vaults with the on-chain `childVaultNonce` sequence. The same applies to the main vault, which is deployed with a fixed nonce of 0.

Recommendation: Add the `childVaultNonce` value used for the deployment to the `ChildVaultCreated` event (e.g. `ChildVaultCreated(child, label, trader, nonce)`) so the salt and address derivation are fully reconstructable from logs.

MNX: Fixed in [ba51f97](#). `ChildVaultCreated` now includes the deployment nonce, and contract tests verify the emitted value.

Researcher: Verified fix.

6.4.18 Missing event emission in bootstrapVaultAccounts

Context: [LiquidityVault.sol#L218](#)

Description: bootstrapVaultAccounts performs critical, one-time initialization - deploying the main vault system account (mainVaultAddress), registering it with the MarginBank, and flipping the irreversible bootstrapped flag but emits no event for these state changes.

Recommendation: Emit a dedicated event (e.g. VaultBootstrapped(mainVaultAddress, initialChild)) at the end of bootstrapVaultAccounts capturing the main vault address and bootstrap completion.

MNX: Fixed in [ba51f97](#). bootstrapVaultAccounts() now emits VaultBootstrapped(mainVault, initialChild), and the contract tests verify the emitted addresses. The corresponding shared application binary interface update is in exchange-monorepo commit 8c3340034.

Researcher: Verified fix.

6.4.19 Unused imports IPerpetual and IGuardian in MarginBank

Context: [MarginBank.sol#L18-L19](#)

Description: Two imports IPerpetual and IGuardian in the MarginBank contract remains unused.

Recommendation: Consider removing the unused imports.

MNX: Fixed in [33bd129](#). Removed the unused IPerpetual and IGuardian imports from MarginBank.sol.

Researcher: Verified fix.

6.4.20 Unnecessary address cast in trustedForwarderOnly modifier

Context: [BaseRelayRecipient.sol#L21](#)

Description: The trustedForwarderOnly modifier casts trustedForwarder to address, which in the first place is of type address. Hence the cast is redundant.

Recommendation: Remove the unnecessary cast:

```
modifier trustedForwarderOnly() {
    require(
-   msg.sender == address(trustedForwarder),
+   msg.sender == trustedForwarder,
        "Function can only be called through the trusted Forwarder"
    );
-;
}
```

MNX: Fixed in [860315a](#). Removed the redundant address(trustedForwarder) cast in the trustedForwarderOnly modifier, trustedForwarder is already an address.

Researcher: Verified fix.

6.4.21

Unused IRelayRecipient import

Context: [BaseRelayRecipient.sol#L4](#)

Description: The imported interface `IRelayRecipient` in `BaseRelayRecipient` is not referenced anywhere and remains unused.

Recommendation: Consider removing the above mentioned unused import.

MNX: Fixed in [860315a](#). Removed the unused `IRelayRecipient` import from `BaseRelayRecipient.sol`.

Researcher: Verified fix.

7 Notes from the Researcher

These closing observations should be read together with the individual findings and remediation statuses recorded in this report.

01 / LIQUIDITY VAULT REMEDIATION

All findings relating to the liquidity vault remain unresolved in the reviewed implementation. The team has indicated that these contracts are undergoing an architectural overhaul; as a result, remediation of the current architecture was not completed during this engagement. The redesigned implementation should be assessed against the outstanding findings, and no liquidity-vault finding should be considered resolved until its replacement has been independently reviewed and verified.

02 / ADDITIONAL ASSURANCE BEFORE SCALING

Given the number and breadth of findings identified during this review, the contracts in their current state should undergo a second independent security audit or a public code competition before the system is scaled or assumes materially greater user or asset exposure. This additional review is recommended to increase assurance; it does not guarantee that the code is free of vulnerabilities.

03 / TEST AND FUZZ COVERAGE

The unit and fuzz test suites should be materially strengthened. In addition to broader line and branch coverage, the test strategy should include integration tests for cross-contract flows, invariant tests for solvency and accounting properties, stateful fuzzing across lifecycle transitions, adversarial boundary conditions, and regression tests for every confirmed finding. The architectural overhaul should be accompanied by a fresh threat model and tests derived from its security invariants.