
Security Review Report

NM-0739 - Palpitada



NETHERMIND
SECURITY

(April 03, 2026)

Contents

1	Executive Summary	3
2	Audited Files	4
3	Summary of Issues	5
4	System Overview	6
4.1	AMM Market	6
4.2	Orderbook Exchange and Market	6
4.3	Multisig Controller	7
4.4	Credit Manager	7
4.5	Governance Token and Fee Discounts	7
4.6	Trust Assumptions and Centralization Risks	8
4.6.1	Market Owner and Protocol Solvency	8
4.6.2	Orderbook Operator	8
4.6.3	Multisig Signers	8
5	Risk Rating Methodology	9
6	Issues	10
6.0.1	[Critical] Buy-side liquidity parameter estimation uses stale <code>liquidityParameter</code> instead of the current dynamic b	10
6.0.2	[High] Credit consumption in <code>predict(...)</code> permanently blocks selling	12
6.0.3	[Medium] Pro-rata <code>usdcSpent</code> deduction on sells enables over-claiming draw refund	13
6.0.4	[Medium] Discrepancy in event date enforcement allows front-running of settlement	14
6.0.5	[Medium] Dynamic fee in direct trades is computed from the taker's limit price instead of the execution price	15
6.0.6	[Medium] Holding period bypass via stale <code>purchaseTimestamp</code> in <code>predict(...)</code>	16
6.0.7	[Medium] Mint/Merge match has no on-chain collateralization enforcement	17
6.0.8	[Medium] OrderbookMarket emergency withdrawal does not update market status	18
6.0.9	[Medium] Ternary draw order can be executed against a different market than the one the maker signed	19
6.0.10	[Low] Cancelled delayed operations can be re-queued by anyone using on-chain signatures	20
6.0.11	[Low] Cancelled orderbook market can be subsequently resolved	21
6.0.12	[Low] Direct trades do not enforce that limit prices cross before execution	22
6.0.13	[Low] Emergency withdrawal does not update market status or prevent subsequent settlement	23
6.0.14	[Low] LMSR math double-scales <code>PRBMath exp()</code> outputs, causing premature overflow in pricing and cost calculations	24
6.0.15	[Low] No signer rotation or removal mechanism after one-time multisig configuration	25
6.0.16	[Low] Sell-generated protocol and curator fees are locked when a market resolves as a draw	25
6.0.17	[Low] Stepwise <code>calculateDynamicB</code> creates price discontinuities at tier boundaries	26
6.0.18	[Low] Switching between manual and dynamic liquidity modes on a live market invalidates pool accounting	26
6.0.19	[Low] Ternary market mint/merge operations hardcode all fees and price improvement to the YES position	27
6.0.20	[Low] <code>sellPosition</code> lacks slippage protection for sellers	27
6.0.21	[Low] <code>setMultisigController</code> in <code>CreditManager</code> does not emit an event	28
6.0.22	[Info] 18-to-6 decimal truncation in <code>sellPosition</code> can burn shares for zero USDC	28
6.0.23	[Info] Curator fees are not subject to token-based fee discounts	29
6.0.24	[Info] EIP-2612 permit flow in <code>predict(...)</code> is broken for users with credit balances	29
6.0.25	[Info] Fee discount can be obtained via flash-loaned tokens	30
6.0.26	[Info] Hardcoded magic numbers used throughout the codebase	31
6.0.27	[Info] Implementation contracts can be initialized by anyone	31
6.0.28	[Info] Inconsistent <code>msg.sender</code> vs <code>_msgSender()</code> usage in <code>CreditManager</code> 's <code>onlyAuthorized</code> modifier	32
6.0.29	[Info] Inconsistent validation for <code>protocolFee</code> between <code>initialize(...)</code> and <code>_setProtocolFee(...)</code>	32
6.0.30	[Info] Market hard-codes a 6-decimal token assumption via <code>DECIMAL_MULTIPLIER</code>	33
6.0.31	[Info] MarketFactory owner storage variable has no setter preventing rotation for market ownership	33
6.0.32	[Info] No combined ceiling on <code>protocolFee</code> + <code>curatorFee</code>	34
6.0.33	[Info] No on-chain order cancellation for regular orders	34
6.0.34	[Info] Non-delayed operations invalidate pending delayed operations via shared nonce	34
6.0.35	[Info] OrderbookMarket emergency withdrawal has no timelock	35
6.0.36	[Info] The <code>executeMultisigOperation(...)</code> function bypasses initialization checks for <code>marketFactory</code>	36
6.0.37	[Info] <code>MAX_SHARES</code> overflow guard is not compatible with the allowed <code>liquidityParameter</code> range	36
6.0.38	[Info] <code>WinnerSet</code> event is misleading when a draw is declared	37
6.0.39	[Info] <code>isRedeemOrderValid</code> returns false for cancelled markets despite <code>executeRedeem</code> succeeding	38
6.0.40	[Info] <code>minimumOrderAmount</code> stored but never enforced	38
6.0.41	[Best Practices] Enum types lack a default sentinel value for uninitialized state	38
6.0.42	[Best Practices] Fee configuration allows setting <code>topicRate</code> to zero after registration	39
6.0.43	[Best Practices] No mechanism to deauthorize markets from <code>CreditManager</code>	39
6.0.44	[Best Practice] Non-standard double-hash EIP-712	40
6.0.45	[Best Practices] OrderbookFactory allows 100% protocol fee at market creation	40
6.0.46	[Best Practices] OrderbookMarket <code>transferFrom</code> silently no-ops for invalid token IDs	40
6.0.47	[Best Practices] State changes in the <code>predict(...)</code> function are ineffective due to an immediate revert	41
6.0.48	[Best Practices] The <code>depositUSDC(...)</code> function is redundant	41
6.0.49	[Best Practice] Wrong error emitted in <code>withdrawCuratorFees</code>	41
6.0.50	[Best Practices] <code>_cancel()</code> in <code>OrderbookMarket</code> does not require <code>ACTIVE</code> status	42
7	Documentation Evaluation	43

8	Test Suite Evaluation	44
8.1	Tests Output	44
8.2	Automated Tools	56
8.2.1	AuditAgent	56
9	About Nethermind	57

1 Executive Summary

This document presents the results of the security review conducted by [Nethermind Security](#) for [Palpitada's](#) prediction market smart contracts. The protocol implements two distinct trading systems: an **AMM-based market** using the Logarithmic Market Scoring Rule (LMSR) for automated price discovery, and an **operator-matched orderbook** for peer-to-peer trading of binary and ternary outcome tokens. Critical operations such as market resolution and emergency withdrawals are governed by a **MultisigController** requiring threshold-based signatures with configurable time delays. The protocol also features a **CreditManager** for promotional credits and a token-based fee discount system. Markets are deployed as minimal proxy clones via factory contracts.

The audit comprises 3621 lines of the Solidity code. The audit was performed using (a) manual analysis of the codebase, and (b) automated analysis tools.

Note

Considering the number of issues found during the audit, we recommend an additional security review before deployment.

Along this document, we report 50 points of attention, where one is classified as Critical, one is classified as High, seven are classified as Medium, twelve are classified as Low, nineteen are classified as Info and ten are classified as Best Practices. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 presents the summary of issues. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tools used. Section 9 concludes the document.

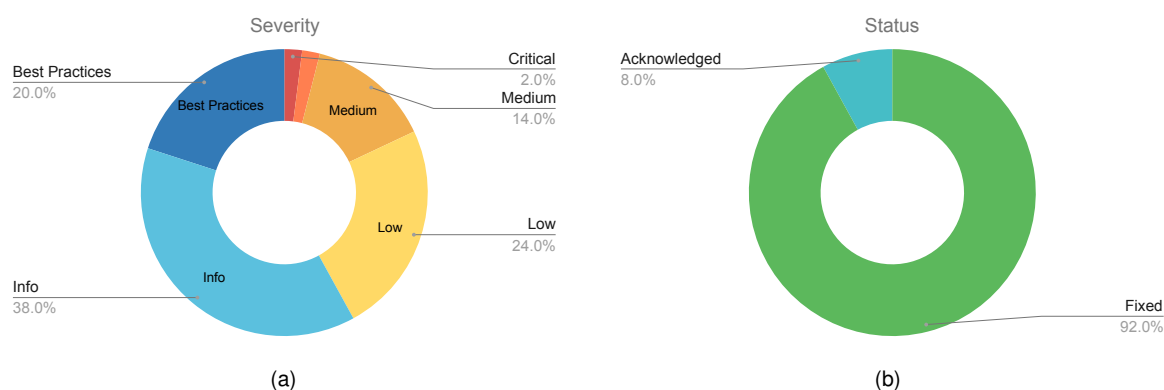


Fig. 1: Distribution of issues: Critical (1), High (1), Medium (7), Low (12), Undetermined (0), Informational (19), Best Practices (10).
Distribution of status: Fixed (46), Acknowledged (4), Mitigated (0), Unresolved (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	March 06, 2026
Final Report	April 03, 2026
Initial Commit	96c9a83
Final Commit	b82429f
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	Multisig/MultisigControllerStorage.sol	13	3	23%	4	20
2	Multisig/MultisigController.sol	198	26	13%	41	260
3	Multisig/factory/MultisigControllerFactoryStorage.sol	3	3	100%	1	7
4	Multisig/factory/MultisigControllerFactory.sol	33	22	66%	8	63
5	Multisig/factory/interfaces/IMultisigFactoryEvents.sol	9	3	33%	1	13
6	Multisig/factory/interfaces/IMultisigFactoryErrors.sol	3	3	100%	1	7
7	Multisig/factory/interfaces/IMultisigControllerFactory.sol	10	9	90%	2	21
8	Multisig/interfaces/IMultisigTarget.sol	4	2	50%	1	7
9	Multisig/interfaces/IMultisigEvents.sol	8	3	37%	1	12
10	Multisig/interfaces/IMultisigController.sol	17	34	200%	9	60
11	Multisig/interfaces/IMultisigErrors.sol	22	3	13%	1	26
12	Multisig/interfaces/IMultisigStructs.sol	19	2	10%	3	24
13	Markets/Market/MarketStorage.sol	60	11	18%	15	86
14	Markets/Market/Market.sol	770	186	24%	159	1112
15	Markets/Market/libraries/MarketCalculations.sol	192	51	26%	32	273
16	Markets/Market/interfaces/IMarketEnums.sol	22	2	9%	3	27
17	Markets/Market/interfaces/IMarketEvents.sol	45	10	22%	11	66
18	Markets/Market/interfaces/IMarketErrors.sol	60	11	18%	10	81
19	Markets/Market/interfaces/IMarketStructs.sol	46	2	4%	7	55
20	Markets/Market/interfaces/IMarket.sol	72	91	126%	21	184
21	Markets/Factory/MarketFactory.sol	110	14	12%	15	139
22	Markets/Factory/MarketFactoryStorage.sol	7	3	42%	3	13
23	Markets/Factory/interfaces/IMarketFactoryEvents.sol	16	2	12%	2	20
24	Markets/Factory/interfaces/IMarketFactoryErrors.sol	5	2	40%	1	8
25	Markets/Factory/interfaces/IMarketFactory.sol	21	19	90%	3	43
26	Governance/GovernanceToken.sol	25	2	8%	2	29
27	Governance/interfaces/IGovernanceTokenEvents.sol	4	2	50%	1	7
28	Governance/interfaces/IGovernanceToken.sol	2	2	100%	1	5
29	Governance/interfaces/IGovernanceTokenErrors.sol	4	2	50%	1	7
30	Orderbook/Token/OutcomeToken.sol	26	25	96%	8	59
31	Orderbook/Exchange/OrderbookExchange.sol	734	276	37%	166	1176
32	Orderbook/Exchange/OrderbookExchangeStorage.sol	43	58	134%	35	136
33	Orderbook/Exchange/libraries/OrderbookFeeCalculations.sol	20	23	115%	7	50
34	Orderbook/Exchange/interfaces/IOrderbookExchangeEvents.sol	56	69	123%	20	145
35	Orderbook/Exchange/interfaces/IOrderbookExchangeStructs.sol	44	29	65%	8	81
36	Orderbook/Exchange/interfaces/IOrderbookExchangeEnums.sol	10	6	60%	3	19
37	Orderbook/Exchange/interfaces/IOrderbookExchangeErrors.sol	38	16	42%	11	65
38	Orderbook/Exchange/interfaces/IOrderbookExchange.sol	35	96	274%	23	154
39	Orderbook/Market/OrderbookMarket.sol	360	97	26%	91	548
40	Orderbook/Market/OrderbookMarketStorage.sol	37	43	116%	38	118
41	Orderbook/Market/interfaces/IOrderbookMarketEvents.sol	28	52	185%	13	93
42	Orderbook/Market/interfaces/IOrderbookMarket.sol	51	74	145%	17	142
43	Orderbook/Market/interfaces/IOrderbookMarketErrors.sol	23	11	47%	7	41
44	Orderbook/Market/interfaces/IOrderbookMarketEnums.sol	7	4	57%	1	12
45	Orderbook/Market/interfaces/IOrderbookMarketStructs.sol	13	10	76%	4	27
46	Orderbook/Factory/OrderbookFactory.sol	116	40	34%	27	183
47	Orderbook/Factory/OrderbookFactoryStorage.sol	9	11	122%	6	26
48	Orderbook/Factory/interfaces/IOrderbookFactoryErrors.sol	10	5	50%	1	16
49	Orderbook/Factory/interfaces/IOrderbookFactoryEvents.sol	16	17	106%	3	36
50	Orderbook/Factory/interfaces/IOrderbookFactory.sol	24	42	175%	12	78
51	Credits/CreditManagerStorage.sol	7	2	28%	1	10
52	Credits/CreditManager.sol	88	26	29%	20	134
53	Credits/interfaces/ICreditManagerErrors.sol	10	13	130%	7	30
54	Credits/interfaces/ICreditManagerEvents.sol	7	2	28%	1	10
55	Credits/interfaces/ICreditManager.sol	9	10	111%	6	25
	Total	3621	1582	43.7%	896	6089

3 Summary of Issues

	Finding	Severity	Update
1	Buy-side liquidity parameter estimation uses stale liquidityParameter instead of the current dynamic b	Critical	Fixed
2	Credit consumption in predict(...) permanently blocks selling	High	Fixed
3	Pro-rata usdcSpent deduction on sells enables over-claiming draw refund	Medium	Fixed
4	Discrepancy in event date enforcement allows front-running of settlement	Medium	Fixed
5	Dynamic fee in direct trades is computed from the taker's limit price instead of the execution price	Medium	Fixed
6	Holding period bypass via stale purchaseTimestamp in predict(...)	Medium	Fixed
7	Mint/Merge match has no on-chain collateralization enforcement	Medium	Fixed
8	OrderbookMarket emergency withdrawal does not update market status	Medium	Fixed
9	Ternary draw order can be executed against a different market than the one the maker signed	Medium	Fixed
10	Cancelled delayed operations can be re-queued by anyone using on-chain signatures	Low	Fixed
11	Cancelled orderbook market can be subsequently resolved	Low	Fixed
12	Direct trades do not enforce that limit prices cross before execution	Low	Fixed
13	Emergency withdrawal does not update market status or prevent subsequent settlement	Low	Fixed
14	LMSR math double-scales PRBMath exp() outputs, causing premature overflow in pricing and cost calculations	Low	Fixed
15	No signer rotation or removal mechanism after one-time multisig configuration	Low	Fixed
16	Sell-generated protocol and curator fees are locked when a market resolves as a draw	Low	Fixed
17	Stepwise calculateDynamicB creates price discontinuities at tier boundaries	Low	Fixed
18	Switching between manual and dynamic liquidity modes on a live market invalidates pool accounting	Low	Fixed
19	Ternary market mint/merge operations hardcode all fees and price improvement to the YES position	Low	Fixed
20	sellPosition lacks slippage protection for sellers	Low	Fixed
21	setMultisigController in CreditManager does not emit an event	Low	Fixed
22	18-to-6 decimal truncation in sellPosition can burn shares for zero USDC	Info	Fixed
23	Curator fees are not subject to token-based fee discounts	Info	Acknowledged
24	EIP-2612 permit flow in predict(...) is broken for users with credit balances	Info	Fixed
25	Fee discount can be obtained via flash-loaned tokens	Info	Acknowledged
26	Hardcoded magic numbers used throughout the codebase	Info	Fixed
27	Implementation contracts can be initialized by anyone	Info	Fixed
28	Inconsistent msg.sender vs _msgSender() usage in CreditManager's onlyAuthorized modifier	Info	Fixed
29	Inconsistent validation for protocolFee between initialize(...) and _setProtocolFee(...)	Info	Fixed
30	Market hard-codes a 6-decimal token assumption via DECIMAL_MULTIPLIER	Info	Fixed
31	MarketFactory owner storage variable has no setter preventing rotation for market ownership	Info	Fixed
32	No combined ceiling on protocolFee + curatorFee	Info	Fixed
33	No on-chain order cancellation for regular orders	Info	Fixed
34	Non-delayed operations invalidate pending delayed operations via shared nonce	Info	Fixed
35	OrderbookMarket emergency withdrawal has no timelock	Info	Fixed
36	The executeMultisigOperation(...) function bypasses initialization checks for marketFactory	Info	Acknowledged
37	MAX_SHARES overflow guard is not compatible with the allowed liquidityParameter range	Info	Fixed
38	WinnerSet event is misleading when a draw is declared	Info	Fixed
39	isRedeemOrderValid returns false for cancelled markets despite executeRedeem succeeding	Info	Fixed
40	minimumOrderAmount stored but never enforced	Info	Fixed
41	Enum types lack a default sentinel value for uninitialized state	Best Practices	Fixed
42	Fee configuration allows setting topicRate to zero after registration	Best Practices	Fixed
43	No mechanism to deauthorize markets from CreditManager	Best Practices	Fixed
44	Non-standard double-hash EIP-712	Best Practices	Fixed
45	OrderbookFactory allows 100% protocol fee at market creation	Best Practices	Fixed
46	OrderbookMarket transferFrom silently no-ops for invalid token IDs	Best Practices	Fixed
47	State changes in the predict(...) function are ineffective due to an immediate revert	Best Practices	Fixed
48	The depositUSDC(...) function is redundant	Best Practices	Acknowledged
49	Wrong error emitted in withdrawCuratorFees	Best Practices	Fixed
50	_cancel() in OrderbookMarket does not require ACTIVE status	Best Practices	Fixed

4 System Overview

4.1 AMM Market

The **Market** contract implements an AMM for binary prediction markets using the Logarithmic Market Scoring Rule (LMSR). Users call `predict()` to purchase shares on either side of an outcome, with prices determined by the LMSR cost function. The liquidity parameter `b` can be set manually by the multisig or derived dynamically from the pool size, which selects `b`. Markets support an early sell feature through `sellPosition()`. Upon resolution, the multisig calls `setWinner()` to settle the market as a win or draw, after which users call `withdraw()` to claim payouts. Draw settlements refund users based on their recorded `usdcSpent`, while winner settlements distribute the `predictionsPoolBalance` proportionally to winning shareholders. Markets are deployed as clones via **MarketFactory**, which also sets up the associated multisig controller and authorizes the market in the credit system, as illustrated in Fig. 2.

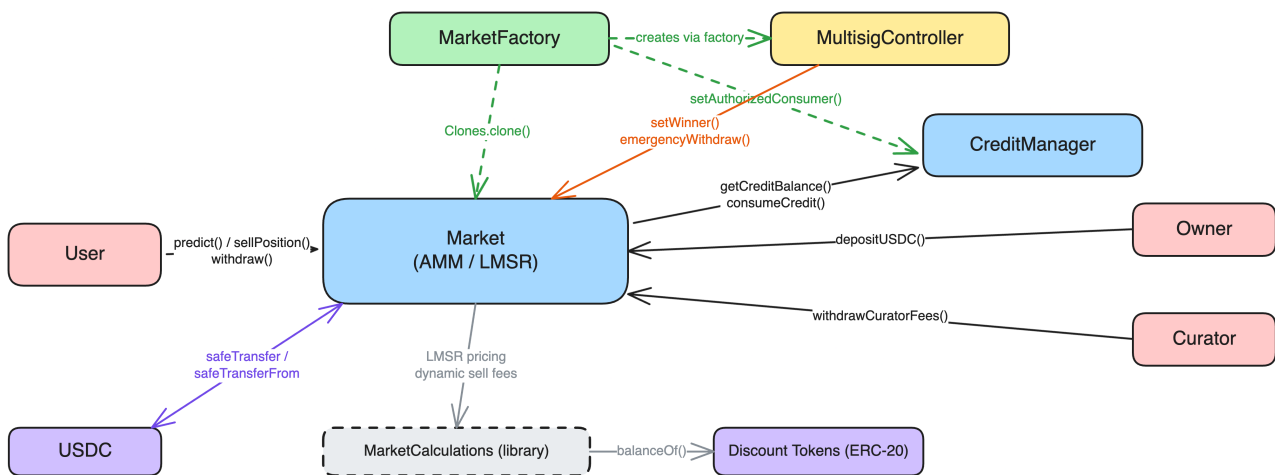


Fig. 2: Operational Flow of the LMSR-Based AMM Market)

4.2 Orderbook Exchange and Market

The **OrderbookExchange** implements an operator-matched orderbook for trading outcome tokens. Users sign orders off-chain, and a trusted operator submits matched pairs via `fillOrders()`. The exchange supports three match types: **DIRECT** trades (token transfers between a buyer and seller), **MINT** operations (matched buyers deposit USDC to mint a full token set), and **MERGE** operations (matched sellers burn a full token set to recover USDC). Both binary (YES/NO) and ternary (YES/NO/DRAW) markets are supported, with ternary operations requiring a third draw order. Fees are computed via a parabolic curve and charged only to the taker. The **OrderbookMarket** contract manages the underlying outcome tokens, which are standard ERC-20 contracts deployed via **OutcomeToken**. It handles minting, burning, token transfers, market resolution, cancellation, and redemption. When a market is cancelled, collateral is distributed to token holders at a flat rate (\$0.50 per token in binary markets, ~\$0.33 in ternary markets) rather than at each user's purchase price. Users who purchased tokens above the cancellation rate incur a net loss while those who purchased below it realize a gain. Markets are deployed through **OrderbookFactory**, which clones the market implementation and registers it with the exchange. Fig. 3 illustrates the end-to-end operational flow of the orderbook system and its interaction with the associated contracts.

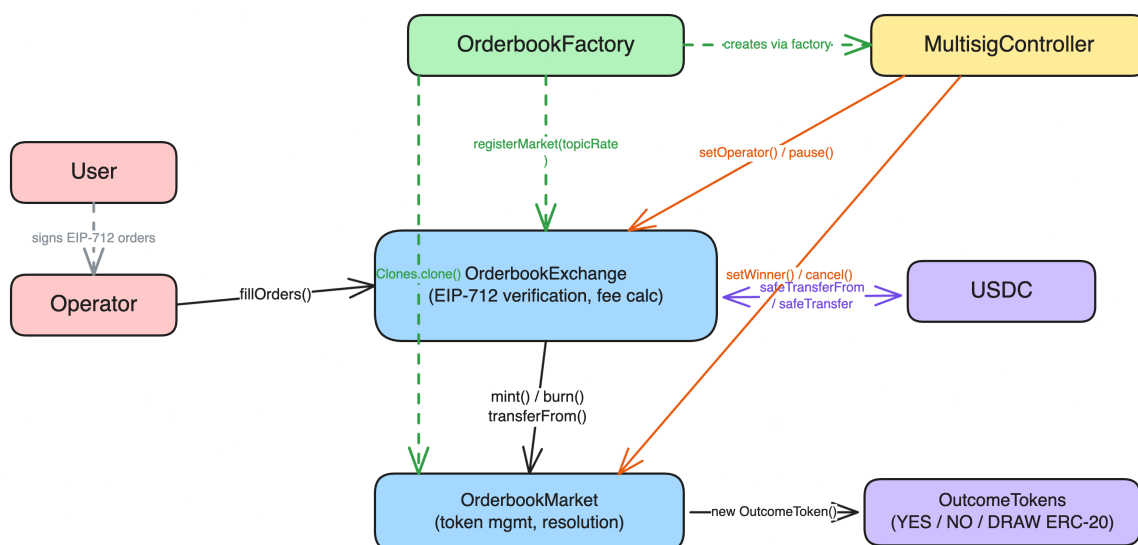


Fig. 3: Operational Flow of the Orderbook System (Exchange, Market, and Multisig Governance)

4.3 Multisig Controller

The **MultisigController** governs critical operations across both market systems. It implements a threshold-based signature scheme where a configurable number of signers must authorize each operation. Configuration is one-time per target via `configureMultisig()`, which sets the signer set, threshold, and per-function delay requirements. Operations are submitted through `executeWithMultisig()` with an array of EIP-712 signatures. Functions configured with a delay are first queued and can only execute after the delay period elapses. Protected functions include market resolution, emergency withdrawals, protocol fee updates, pause and unpaused controls, and operator or fee configuration changes. Each AMM market, each orderbook market, the orderbook exchange, and the credit manager receive their own **MultisigController** instance, with per-market instances deployed via **MultisigControllerFactory**, as illustrated in Fig. 4.

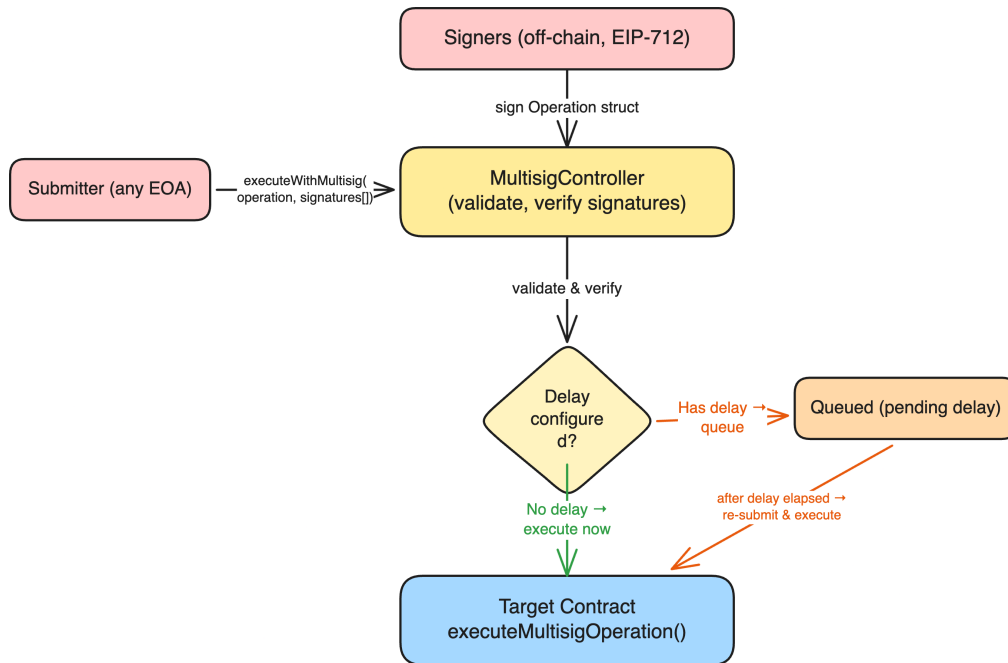


Fig. 4: MultisigController Execution Flow and Delayed Operation Mechanism

4.4 Credit Manager

The **CreditManager** allows the protocol to issue promotional credits to users, which are automatically consumed during predictions in the AMM market. Credits are denominated in 6-decimal USDC precision and are deducted from the total cost before any USDC transfer occurs in `predict()`. The contract tracks per-user balances and exposes `addCredit()` and `consumeCredit()` functions restricted to authorized consumers (markets) and the contract owner. Market contracts are authorized as consumers when created by the factory via `setAuthorizedConsumer()`. The credit manager also supports multisig-governed operations for adding credits and updating the market factory address.

4.5 Governance Token and Fee Discounts

The **GovernanceToken** (PPT) is a standard ERC-20 token with ERC-2612 permit support. It serves as one of potentially multiple discount tokens that can be configured per market. The AMM market supports a tiered fee discount system where holders of configured ERC-20 tokens receive basis-point reductions on protocol fees. Discount configurations are managed per-market via the multisig through `setTokenDiscountConfig()`, which defines balance thresholds and corresponding discount tiers. The discount is applied at prediction time by checking the user's token balance and selecting the highest applicable tier.

4.6 Trust Assumptions and Centralization Risks

The protocol relies on several trusted roles whose correct and timely behavior is essential for the system to function as intended. The following outlines the key trust dependencies and the risks they introduce.

4.6.1 Market Owner and Protocol Solvency

The AMM market contract does not guarantee that its USDC balance is always sufficient to cover all outstanding obligations. Two scenarios create a gap between the contract's USDC holdings and its liabilities:

- a. **Profitable early sales.** When a user buys shares and later sells at a higher LMSR price, the contract pays out more USDC than the user's proportional `usdcSpent` deduction. In a draw settlement, the remaining users' refund claims may exceed the contract's balance.
- b. **Credit-funded predictions.** Credits consumed via `CreditManager` are added to `predictionsPoolBalance` without a corresponding USDC deposit. Winner payouts are computed as a proportional share of `predictionsPoolBalance`, meaning the contract must hold additional USDC to honor credit-inflated pool obligations.

In both cases, the protocol owner is expected to monitor the deficit and deposit USDC through `depositUSDC()` before settlement. If the owner fails to do so, the existing `InsufficientContractFunds` check in `withdraw()` will cause the last claimants' transactions to revert rather than paying out unbacked funds. While this prevents insolvency, it shifts the solvency guarantee from on-chain enforcement to an off-chain operational responsibility.

4.6.2 Orderbook Operator

The `OrderbookExchange` delegates all order matching to a single trusted operator. Users sign EIP-712 orders off-chain, but the operator alone decides which orders to match, when to submit them, and in what combination. This introduces several trust assumptions:

- The operator can selectively delay or withhold order submissions without on-chain accountability.
- The operator sets the `takerDiscountBps` parameter per fill, controlling the effective fee discount applied to each trade.
- For MINT and MERGE operations, the operator selects which orders to group, influencing price improvement allocation.
- Single-user operations (`executeSplit`, `executeMerge`, `executeRedeem`) also require operator submission, meaning users cannot independently interact with the exchange contract.

If the operator becomes unavailable or acts maliciously, users have no on-chain mechanism to cancel pending orders, force execution, or bypass the operator. Signed orders remain valid until their expiration, but users cannot revoke them except by incrementing their nonce through a future fill.

4.6.3 Multisig Signers

Critical operations across both market systems are governed by `MultisigController` instances with threshold-based signature requirements. The multisig controls market resolution (`setWinner`), emergency fund withdrawals, protocol fee changes, and market lifecycle transitions. An incorrect or malicious resolution directly determines the distribution of all funds held in the market. The signer set is configured once per target and cannot be rotated, meaning a compromised or lost key permanently degrades the effective threshold. For functions without a configured delay, execution is immediate upon reaching the signature threshold, with no timelock or on-chain dispute window.

5 Risk Rating Methodology

The risk rating methodology used by [Nethermind Security](#) follows the principles established by the [OWASP Foundation](#). The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

- a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;
- b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;
- c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

- a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;
- b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;
- c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	Medium	High
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, [Nethermind Security](#) also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

- a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;
- b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;
- c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.0.1 [Critical] Buy-side liquidity parameter estimation uses stale liquidityParameter instead of the current dynamic b

File(s): Markets/Market/Market.sol

Description: When autoDynamicLiquidity is enabled, the buy path and sell path use different methods to determine the effective b parameter. The sell path correctly derives b from the actual pool size:

```
function getEffectiveLiquidityParameter() public view override returns (uint256) {
    if (autoDynamicLiquidity) {
        uint256 totalPoolSize = totalCostA + totalCostB;
        return MarketCalculations.calculateDynamicB(totalPoolSize);
    }
    return liquidityParameter;
}
```

The buy path, however, attempts to project the post-trade pool size to select b. It estimates the cost of the incoming purchase using the stored liquidityParameter rather than the current dynamic b:

```
function _getEffectiveBForPurchase(uint256 sharesToBuy, bool onSideA) internal view returns (uint256) {
    if (!autoDynamicLiquidity) {
        return liquidityParameter;
    }
    uint256 currentB = liquidityParameter > 0 ? liquidityParameter : DEFAULT_LIQUIDITY_PARAMETER;
    uint256 currentPrice = MarketCalculations.getPrice(sharesA, sharesB, onSideA, currentB);
    uint256 estimatedCost = (sharesToBuy * currentPrice);

    uint256 currentPoolSize = totalCostA + totalCostB;
    uint256 projectedPoolSize = currentPoolSize + estimatedCost;
    return MarketCalculations.calculateDynamicB(projectedPoolSize);
}
```

The liquidityParameter is the value set at market creation. Once the pool crosses a tier boundary (e.g., past \$1,000), getEffectiveLiquidityParameter() returns a larger b (e.g., 10,000), but _getEffectiveBForPurchase still uses the original liquidityParameter (e.g., 1,000) for its price estimate.

A smaller b produces a steeper LMSR price curve, which pushes the favored side's marginal price toward 1.0. This inflates estimatedCost, which inflates projectedPoolSize, which causes calculateDynamicB to return a b from a higher tier than what the actual post-trade pool size would justify. The buy is then priced under this inflated b via calculateAmountRequiredInUSDC(...):

```
function calculateAmountRequiredInUSDC(uint256 sharesToBuy, bool onSideA)
    public
    view
    override
    returns (uint256, uint256, uint256, uint256)
{
    uint256 effectiveB = _getEffectiveBForPurchase(sharesToBuy, onSideA);

    uint256 initialCost = MarketCalculations.getCostFunction(sharesA, sharesB, effectiveB);
    uint256 newCost;

    if (onSideA) {
        newCost = MarketCalculations.getCostFunction(sharesA + sharesToBuy, sharesB, effectiveB);
    } else {
        newCost = MarketCalculations.getCostFunction(sharesA, sharesB + sharesToBuy, effectiveB);
    }

    uint256 cost = newCost - initialCost;
    // ...
}
```

A larger b means a flatter cost curve and lower price impact, so the buyer pays less for the same number of shares. After the trade, the actual pool grows by this lower real cost rather than the inflated estimate, so getEffectiveLiquidityParameter() still returns the lower-tier b. When the buyer then sells, the return is computed under this correct, smaller b, which values each share higher.

In an example scenario, a market with liquidityParameter = 1,000 and autoDynamicLiquidity = true has grown to a pool size of 4,349 through normal 70/30 trading activity. The actual effective b from getEffectiveLiquidityParameter() is 10,000, but the stored liquidityParameter remains 1,000.

An attacker buys 6,000 shares on the favored side:

1. `_getEffectiveBForPurchase` uses `currentB = 1,000` (stale). Under this small `b`, the price for the favored side is 0.985 (near 1.0). The correct price under `b = 10,000` would be 0.54;
2. `estimatedCost = 6,000 * 0.985 = 5,910`. The correct estimate would be $6,000 * 0.54 = 3,240$;
3. `projectedPoolSize = 4,349 + 5,910 = 10,259`, which crosses the \$10K tier boundary, so the buy is priced under `b = 100,000`. The correct projection ($4,349 + 3,240 = 7,589$) would stay in the 1K–10K tier at `b = 10,000`;
4. Under `b = 100,000`, the LMSR cost curve is very flat and the attacker pays only **\$3,108** for 6,000 shares;
5. The actual pool grows to \$7,458 (still in the \$1K-\$10K tier, `b = 10,000`);
6. The attacker sells immediately. The sell path uses the real `b = 10,000` and returns **\$4,029** before fees;
7. After the 1% sell fee, the attacker nets **\$881 profit**, extracted directly from other users' collateral in the pool;

Recommendation(s): Consider updating the logic to use the `getEffectiveLiquidityParameter()` in `_getEffectiveBForPurchase(...)`

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.2 [High] Credit consumption in predict(...) permanently blocks selling

File(s): Markets/Market/Market.sol

Description: The predict(...) function automatically consumes any available credits from the user's CreditManager balance before processing a prediction. This consumption is greedy it applies as much credit as possible and there is no parameter or mechanism for a user to opt out.

```
function predict(
    bool onSideA,
    uint256 sharesToBuy,
    uint256 maxPricePerShare,
    PermitData callData permit
) external override nonReentrant returns (uint256) {
    // ...
    uint256 creditsUsed = 0;
    uint256 userCredits = creditManager.getCreditBalance(msg.sender);

    if (userCredits > 0) {
        creditsUsed = Math.min(totalAmountRequiredInUSDC, userCredits);
    }

    uint256 usdcAmountRequired = totalAmountRequiredInUSDC - creditsUsed;
    // ...
    position.creditsUsed += creditsUsed;
    // ...
}
```

Once any amount of credits is recorded on a position, sellPosition(...) permanently rejects sells for that side:

```
function sellPosition(bool fromSideA, uint256 sharesToSell) external override nonReentrant returns (uint256) {
    // ...
    UserPosition storage position = positions[msg.sender][fromSideA];
    if (position.creditsUsed != 0) {
        revert CannotSellCreditsPosition();
    }
    // ...
}
```

The combination of these two behaviors creates a situation where a user who holds any credit balance no matter how small will have that credit silently consumed on their next prediction, permanently locking that entire side's position from being sold. The user's only exit becomes waiting for settlement (win or draw).

This is particularly problematic in the following scenarios:

- The protocol distributes small promotional credits (e.g., \$1) to attract users. A user who then makes a large USDC-funded prediction (e.g., \$50) has the \$1 credit automatically applied, making the entire \$50 position unsellable despite being 98% USDC-funded;
- When a market settles as a draw, withdraw(...) returns the user's previously consumed credits via creditManager.addCredit(...). If the user later participates in a new market, those restored credits are automatically consumed, once again locking their new position from selling;
- The sell restriction is binary (creditsUsed != 0) rather than proportional. There is no threshold or ratio check a single wei of credits on a position worth thousands of dollars in USDC is treated identically to a fully credit-funded position;

Recommendation(s): Consider scaling the sell restriction proportionally to the credit-funded portion of a position rather than applying a blanket block on any non-zero credit usage.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.3 [Medium] Pro-rata usdcSpent deduction on sells enables over-claiming draw refund

File(s): Markets/Market/Market.sol

Description: On a draw, `withdraw(...)` refunds users based on `position.usdcSpent`:

```
function withdraw() external override nonReentrant {
    // ...
    if (isDraw) {
        uint256 totalUsdcSpent = positions[msg.sender][true].usdcSpent + positions[msg.sender][false].usdcSpent;
        payout = totalUsdcSpent - creditsToBeReturned;
        // ...
    }
    // ...
}
```

When a user sells before settlement, `usdcSpent` is reduced pro-rata by shares:

```
function sellPosition(bool fromSideA, uint256 sharesToSell) external override nonReentrant returns (uint256) {
    // ...
    uint256 usdcToDeduct = (position.usdcSpent * sharesToSell) / userShares;
    position.usdcSpent -= usdcToDeduct;
    // ...
}
```

This deduction is based on the user's historical average cost per share, not the actual USDC the AMM returned for those shares. When the AMM returns more than the average cost (a profitable sell), `usdcSpent` is only partially reduced while the full sell payout leaves the contract. This leaves `usdcSpent` overstated for the remaining position.

Consider the following scenario:

1. Alice buys 50 shares on Side A for 25 USDC. `usdcSpent` = 25;
2. Bob buys 200 shares on Side A, pushing the price up;
3. Alice sells 25 shares. The AMM returns 12.72 USDC, but the pro-rata deduction is only $25 * 25/50 = 12.50$. Her remaining `usdcSpent` = 12.50;
4. The contract paid out 12.72 but only reduced the draw liability by 12.50, creating a 0.22 USDC gap;
5. Draw is declared. Alice claims 12.50 as a draw refund;
6. Alice received 12.72 (sell) + 12.50 (draw) = 25.22 USDC total, despite depositing 25;

The 0.22 USDC shortfall falls on the last user to call `withdraw(...)`, whose transaction reverts with `InsufficientContractFunds`. This is not balanced by other users' deposits because each buyer's USDC is fully spoken for by their own `usdcSpent` draw claim.

Recommendation(s): Consider tracking the cumulative deficit between AMM sell proceeds and pro-rata `usdcSpent` deductions in a state variable whenever the actual payout diverges from the deducted amount. This variable should be exposed via a view function so the protocol can monitor the gap and deposit USDC via before draw settlement.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.4 [Medium] Discrepancy in event date enforcement allows front-running of settlement

File(s): Markets/Market/Market.sol

Description: The Market contract contains a logic discrepancy in how the eventDate is enforced between the predict(...) and _setWinner(...) functions.

The predict(...) function only prevents users from placing new predictions when the current block.timestamp is strictly greater than the eventDate. Consequently, if the block.timestamp is exactly equal to the eventDate, a purchase is still permitted.

```
function predict(...) external override nonReentrant returns (uint256) {
    // ...
    // solhint-disable-next-line not-rely-on-time
    // @audit-issue Strict inequality allows purchases when block.timestamp == eventDate.
    if (block.timestamp > eventDate) {
        // ...
        revert EventClosed();
    }
    // ...
}
```

Conversely, the _setWinner(...) function allows the market to be transitioned from OPEN to CLOSED and settled as soon as the block.timestamp is greater than or equal to the eventDate.

```
function _setWinner(bool sideAHasWon_, bool draw) internal {
    // ...
    if (marketStatus == MarketStatus.OPEN) {
        // solhint-disable-next-line not-rely-on-time
        // @audit-issue Settlement is allowed when block.timestamp == eventDate.
        if (block.timestamp >= eventDate) {
            marketStatus = MarketStatus.CLOSED;
            emit MarketStatusChanged(MarketStatus.OPEN, MarketStatus.CLOSED);
        } else {
            revert EventNotClosed();
        }
    }
    // ...
}
```

This creates a one-block window where a market is both open for trading and eligible for settlement. An attacker can monitor the mempool for a settlement transaction (which reveals the winner in its calldata) and front-run it with a large predict(...) transaction. Because the attacker places the trade at the exact moment the outcome is revealed or settlement is possible, they can acquire a massive portion of the winning side's shares.

Recommendation(s): Consider synchronizing the time boundary across both functions.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.5 [Medium] Dynamic fee in direct trades is computed from the taker's limit price instead of the execution price

File(s): Orderbook/Exchange/OrderbookExchange.sol, Orderbook/Exchange/libraries/OrderbookFeeCalculations.sol

Description: The dynamic fee in `_executeDirect` is meant to reflect the risk profile of a trade. The parabolic fee curve `topicRate * price * (1 - price)` charges more for trades near 0.50 (uncertain outcomes) and less for trades near 0.00 or 1.00 (near-certain outcomes). This makes sense when the price fed into the curve represents where the trade actually executed. However, the function uses the taker's limit price rather than the execution price:

```
function _executeDirect(
    IOrderbookMarket market,
    Order calldata takerOrder,
    Order calldata makerOrder,
    uint256 fillAmount,
    uint256 takerDiscountBps
) internal {
    // ...
    uint256 takerPrice = (takerOrder.takerAmount * 1e18) / takerOrder.makerAmount;
    uint256 makerPrice = (makerOrder.takerAmount * 1e18) / makerOrder.makerAmount;

    if (takerOrder.side == Side.BUY) {
        uint256 executionPrice = makerPrice < takerPrice ? makerPrice : takerPrice;
        usdcAmount = (fillAmount * executionPrice) / 1e18;
    } else {
        uint256 executionPrice = makerPrice > takerPrice ? makerPrice : takerPrice;
        usdcAmount = (fillAmount * executionPrice) / 1e18;
    }

    uint256 price = _extractPrice(takerOrder);
    (uint256 takerFee, uint256 baseFeeRate) = _calculateDynamicFee(
        address(market),
        usdcAmount,
        price,
        takerDiscountBps
    );
    // ...
}
```

`_extractPrice` returns the taker's own limit ratio, not the execution price:

```
function _extractPrice(Order calldata order) internal pure returns (uint256) {
    if (order.makerAmount == 0) return 0;
    return (order.takerAmount * PRICE_PRECISION) / order.makerAmount; // ...
}
```

The limit price represents the worst price the taker is willing to accept, not the price at which the trade settles. Due to price improvement, the execution price can differ from the limit. A taker who sets a limit at \$0.90 might execute at \$0.55 through price improvement. The fee should reflect the \$0.55 execution since that is the actual risk position being taken, but instead it reflects the \$0.90 limit. At \$0.90, the fee curve produces $\text{topicRate} * 0.90 * 0.10 = 0.09 * \text{topicRate}$, whereas the correct fee at \$0.55 would be $\text{topicRate} * 0.55 * 0.45 = 0.2475 * \text{topicRate}$.

This becomes more pronounced at extremes. A limit near \$1.00 produces a fee approaching zero from the curve ($\text{topicRate} * 0.99 * 0.01$), while the actual execution could be anywhere. The minimum fee floor does not help because it only applies when $\text{fee} > 0$:

```
if (fee > 0 && fee < minimumFeeAmount) {
    fee = minimumFeeAmount;
}
```

When integer arithmetic rounds the parabolic result to exactly zero, the floor is bypassed entirely.

The result is that fees do not correspond to the actual trade that occurred. Two identical trades at the same execution price can produce different fee amounts depending on how each taker set their limit, which is not a meaningful factor in the risk of the position.

Recommendation(s): Consider computing the fee using the actual execution price rather than the taker's limit price.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.6 [Medium] Holding period bypass via stale purchaseTimestamp in predict(...)

File(s): Markets/Market/Market.sol

Description: The predict(...) function is the primary entry point for users to buy shares in a market. When a user makes their first prediction on a given side, the contract records the current block.timestamp in position.purchaseTimestamp. This timestamp is later used by sellPosition(...) to enforce the minHoldingPeriod a configurable delay (e.g., 1 hour) that prevents users from immediately flipping positions.

However, purchaseTimestamp is only written on the first purchase for a given side, and is never updated on subsequent buys:

```
function predict(
    bool onSideA,
    uint256 sharesToBuy,
    uint256 maxPricePerShare,
    PermitData calldata permit
) external override nonReentrant returns (uint256) {
    // ...
    UserPosition storage position = positions[msg.sender][onSideA];
    position.shares += sharesToBuy;
    position.usdcSpent += totalAmountRequiredInUSDC;
    position.creditsUsed += creditsUsed;
    if (position.purchaseTimestamp == 0) {
        position.purchaseTimestamp = block.timestamp;
    }
    // ...
}
```

The holding period check in sellPosition(...) then uses this stale timestamp:

```
function sellPosition(bool fromSideA, uint256 sharesToSell) external override nonReentrant returns (uint256) {
    // ...
    if (block.timestamp < position.purchaseTimestamp + sellConfig.minHoldingPeriod) {
        revert HoldingPeriodNotMet();
    }
    // ...
}
```

Any user can exploit this by "pre-aging" a position with a negligible dust buy (e.g., 1 share), waiting for the holding period to elapse, and then purchasing an arbitrarily large number of shares. Because the purchaseTimestamp remains anchored to the original dust buy, the user can immediately sell the newly acquired shares, completely bypassing the intended holding period.

Recommendation(s): Consider creating a more granular approach for the holding period such as tracking a weighted-average purchase time or maintaining per-batch timestamps.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.7 [Medium] Mint/Merge match has no on-chain collateralization enforcement

File(s): [Orderbook/Exchange/OrderbookExchange.sol](#)

Description: In `_executeMint`, the code calculates each party's payment (`yesPay`, `noPay`) based on their order prices. If `totalPay > fillAmount`, surplus is returned to the taker as price improvement. However, if `totalPay < fillAmount`, the underpayment is silently accepted. The market's `mint()` function then records `totalUsdcReserve += fillAmount` (not `+= totalPay`), inflating the reserve beyond actual USDC held:

```
uint256 totalPay = yesPay + noPay + drawPay;
if (totalPay > fillAmount) {
    uint256 surplus = totalPay - fillAmount;
    // ... reduce taker's payment
}
// No else branch for totalPay < fillAmount

IERC20(usdc).safeTransferFrom(yesRecipient, address(market), yesPay);
IERC20(usdc).safeTransferFrom(noRecipient, address(market), noPay);

market.mint(yesRecipient, noRecipient, drawRecipient, fillAmount);
```

As an example, Alice BUY 100 YES at \$0.30, Bob BUY 100 NO at \$0.30. `totalPay` = 60 USDC, `fillAmount` = 100. The market receives 60 USDC but records `totalUsdcReserve` = 100. The 40 USDC shortfall means winners cannot fully redeem.

The same issue exists in `_executeMerge` in the opposite direction. If both SELL orders demand prices that sum above \$1.00 per token set, `totalProceeds > fillAmount` is not checked:

```
uint256 totalProceeds = yesProceeds + noProceeds + drawProceeds;
if (totalProceeds < fillAmount) {
    uint256 surplus = fillAmount - totalProceeds;
    // ... increase taker's proceeds
}
// No else branch for totalProceeds > fillAmount

market.burn(yesHolder, noHolder, drawHolder, fillAmount, yesProceeds, noProceeds, drawProceeds, takerFee);
```

As an example, Alice SELL 100 YES at \$0.70, Bob SELL 100 NO at \$0.70. `totalProceeds` = 140 USDC, `fillAmount` = 100. The market's `burn` decrements `totalUsdcReserve` by 100 but transfers 140 USDC out, draining 40 USDC more than the token set is worth.

Both cases lead to the same outcome, the market's actual USDC balance diverges from `totalUsdcReserve`. A compromised or buggy operator can create undercollateralized markets through mint or drain existing collateral through merge, leaving token holders facing insolvency at redemption.

Recommendation(s): Consider adding a check to prevent this scenario.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.8 [Medium] OrderbookMarket emergency withdrawal does not update market status

File(s): Orderbook/Market/OrderbookMarket.sol

Description: `_emergencyWithdraw` drains all USDC from the market and resets `totalUsdcReserve` to zero, but does not update status or resolution:

```
function _emergencyWithdraw(address recipient) internal {
    if (recipient == address(0)) revert ZeroAddress();

    uint256 balance = IERC20(usdc).balanceOf(address(this));
    if (balance == 0) revert ZeroAmount();

    // Reset total reserve
    totalUsdcReserve = 0;

    // Transfer all USDC to recipient
    IERC20(usdc).safeTransfer(recipient, balance);

    emit EmergencyWithdrawal(recipient, balance);
}
```

Since status remains ACTIVE, user-facing operations continue to function on a market with no backing. When `splitMergeRestricted` is false, any user can call `split()` directly to deposit USDC into the drained market. Once fresh USDC enters the contract, old token holders (who received tokens before the drain) can call `merge()` to extract it. For example, if User B deposits 100 USDC via `split()`, User A who held 100 YES + 100 NO tokens from before the drain can immediately `merge()` to claim that 100 USDC. User B is then unable to fully merge their own tokens because the contract no longer holds enough USDC.

Additionally, outcome tokens are standard ERC20 contracts with unrestricted transfers. Old token holders can sell or transfer their unbacked tokens freely to other users who have no on-chain indication that the market's collateral was drained.

Recommendation(s): Consider setting the status to a terminal state inside `_emergencyWithdraw`.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.9 [Medium] Ternary draw order can be executed against a different market than the one the maker signed

File(s): Orderbook/Exchange/OrderbookExchange.sol

Description: Every execution path in the exchange correctly binds signed orders to their intended market except the ternary draw order path. For binary and 2-order matches, `fillOrders(...)` enforces that both parties signed for the same market:

```
if (takerOrder.market != makerOrder.market) revert MarketMismatch();

IOrderbookMarket market = IOrderbookMarket(takerOrder.market);
```

The single-user operations (`executeSplit`, `executeMerge`, `executeRedeem`) are inherently safe because they use the signed order's own market field for both validation and execution, there is no second order whose market could diverge.

The ternary draw order path diverges from both patterns. It introduces a third independently-signed order, but unlike the taker/maker pair, the draw order's market is never compared against the execution market:

```
if (hasDrawOrder) {
    if (!OrderbookMarket(address(market)).hasDraw()) revert NotTernaryMarket();

    drawHash = _validateOrder(drawOrder, address(0), drawSignature);
    _updateOrderFilled(drawHash, drawOrder.makerAmount, fillAmount);

    if (takerOrder.tokenId != 0 || makerOrder.tokenId != 1 || drawOrder.tokenId != 2) {
        revert InvalidTokenIds();
    }

    if (takerOrder.side != makerOrder.side || takerOrder.side != drawOrder.side) {
        revert SideMismatch();
    }
}
```

`_validateOrder(...)` verifies the draw order's EIP-712 signature against `drawOrder.market` (confirming the signer authorized that specific market) and checks that `drawOrder.market` is registered and active. However, these are isolated validations. The draw order passes all its own checks, but is never anchored to the execution context established by the taker/maker pair.

The subsequent execution in `_executeMint(...)` uses the market variable (derived from `takerOrder.market`) for all fund transfers and token minting, including the draw maker's:

```
if (hasDrawOrder && drawPay > 0) {
    IERC20(usdc).safeTransferFrom(drawRecipient, address(market), drawPay);
}

market.mint(yesRecipient, noRecipient, drawRecipient, fillAmount);
```

Because USDC is fungible, the `safeTransferFrom` succeeds regardless of which market the draw maker originally intended. The draw maker's USDC is transferred to `takerOrder.market` (Market B) and DRAW tokens are minted for them in Market B, even though their EIP-712 signature specifically authorized `drawOrder.market` (Market A).

Recommendation(s): Consider adding a check such as `if (drawOrder.market != takerOrder.market) revert MarketMismatch();` inside the `hasDrawOrder` block.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.10 [Low] Cancelled delayed operations can be re-queued by anyone using on-chain signatures

File(s): Multisig/MultisigController.sol

Description: When a delayed operation is first submitted via `executeWithMultisig(...)`, the caller provides the required signer signatures as calldata. These signatures become permanently visible on-chain in the transaction's input data. If the signers later cancel the operation, the cancellation clears the queue state but does not invalidate the signatures:

```
function _cancelOperation(Operation memory operation) internal {
    bytes32 operationHash = _hashOperation(operation);
    if (!operationQueued[operationHash]) revert OperationNotQueued();

    delete pendingOperations[operationHash];
    delete operationQueued[operationHash];
    emit OperationCancelled(operationHash);
}
```

After cancellation, the original operation remains valid because:

1. The target nonce has not advanced. The cancel operation increments `nonces[address(this)]` (the self-target nonce), but the market/target nonce is unchanged;
2. The operation is not marked in `executedOperations`;
3. The original signatures still recover to valid signers for the unchanged operation hash;

Any observer can extract the signatures from the original queuing transaction's calldata and call `executeWithMultisig(operation, signatures)` again, re-queuing the cancelled operation with a fresh delay timer. The signers must cancel again, and this cycle can repeat indefinitely. Each cancellation requires coordinating threshold signatures for a new cancel operation (consuming a self-target nonce), while re-queuing costs only a single transaction using the already-known signatures.

The only way to permanently stop the re-queuing is to execute a different operation for the same target, which advances the target nonce and invalidates the old operation.

Recommendation(s): Consider incrementing the target nonce when an operation is cancelled, or marking the cancelled operation hash in `executedOperations` so it cannot be resubmitted.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.11 [Low] Cancelled orderbook market can be subsequently resolved

File(s): Orderbook/Market/OrderbookMarket.sol

Description: `_cancel()` sets `status = CANCELLED` but does not set `resolution.resolved = true`. A subsequent `_setWinner()` call only checks `resolution.resolved`, not the status. This allows a cancelled market to be resolved, overwriting the cancellation with `status = RESOLVED`.

In the Orderbook system, 1 YES token + 1 NO token together represent \$1.00 of collateral held in the reserve. The payout per token depends entirely on market state at redemption time:

Market state	Winning token payout	Losing token payout
CANCELLED	\$0.50 (50% of face value)	\$0.50 (50% of face value)
RESOLVED	\$1.00 (100% of face value)	\$0.00

The cancellation path in `redeemFor()` explicitly applies a 50% rate:

```
uint256 yesUsdc = (yesAmount * 5000) / 10000; // 50%
uint256 noUsdc  = (noAmount  * 5000) / 10000; // 50%
```

The resolution path pays 1 token = 1 USDC to the winning side only.

An example vulnerable sequence is as follows:

1. Multisig calls `_cancel()`. Status becomes CANCELLED, but `resolution.resolved` stays false;
2. Alice sees CANCELLED and redeems immediately. She holds 100 YES tokens and receives 50 USDC ($\$0.50 \times 100$);
3. Bob holds 100 YES tokens and waits;
4. Multisig calls `_setWinner()`. Because `resolution.resolved` is still false, the guard at line 440 passes. The market is now RESOLVED with YES as winner, overwriting the cancelled state;
5. Bob calls `redeemFor()`. He holds 100 YES tokens and receives 100 USDC ($\$1.00 \times 100$);

Resulting inequality is shown below:

User	Tokens	Redeemed when	Received	Per token
Alice	100 YES	During CANCELLED	50 USDC	\$0.50
Bob	100 YES	After RESOLVED	100 USDC	\$1.00

Same tokens, same market, same side. Alice receives half of what Bob receives purely because she acted in good faith on the state the contract publicly advertised. The bug lets the protocol retroactively change the payout rules after users have already redeemed under the prior state.

Recommendation(s): Consider adding `if (status == OrderbookMarketStatus.CANCELLED) revert MarketCancelled();` in `_setWinner()`.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.12 [Low] Direct trades do not enforce that limit prices cross before execution

File(s): Orderbook/Exchange/OrderbookExchange.sol

Description: In `_executeDirect`, the contract computes an execution price using a price improvement rule (min for buy taker, max for sell taker) but never verifies that the two orders' limit prices are compatible:

```
function _executeDirect(
    IOrderbookMarket market,
    Order calldata takerOrder,
    Order calldata makerOrder,
    uint256 fillAmount,
    uint256 takerDiscountBps
) internal {
    // ...
    uint256 takerPrice = (takerOrder.takerAmount * 1e18) / takerOrder.makerAmount;
    uint256 makerPrice = (makerOrder.takerAmount * 1e18) / makerOrder.makerAmount;

    if (takerOrder.side == Side.BUY) {
        // ...
        uint256 executionPrice = makerPrice < takerPrice ? makerPrice : takerPrice;
        usdcAmount = (fillAmount * executionPrice) / 1e18;
    } else {
        // ...
        uint256 executionPrice = makerPrice > takerPrice ? makerPrice : takerPrice;
        usdcAmount = (fillAmount * executionPrice) / 1e18;
    }

    IERC20(usdc).safeTransferFrom(buyer, seller, usdcAmount);
    market.transferFrom(seller, buyer, fillAmount, tokenId);
}
```

For a buy taker, the execution price is $\min(\text{makerPrice}, \text{takerPrice})$. If the buyer's limit is \$0.40 and the seller's limit is \$0.60, the orders do not cross (the buyer won't pay enough to meet the seller's minimum). However, the code does not reject this match. It sets $\text{executionPrice} = \$0.40$ and executes, transferring \$0.40 from the buyer and forcibly moving the seller's tokens via `market.transferFrom`. The seller receives less than the \$0.60 minimum they signed for.

For a sell taker, the reverse occurs: $\text{executionPrice} = \max(\text{makerPrice}, \text{takerPrice})$ can force the buyer to pay above their signed limit.

Signature validation confirms that both parties signed orders with the given fields (market, amounts, token, side), but the contract never checks that the two orders' prices are mutually satisfiable. The operator is trusted to only match crossing orders, but the on-chain contract provides no enforcement.

Recommendation(s): Consider adding a crossing price check in `_executeDirect` before computing the execution price.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.13 [Low] Emergency withdrawal does not update market status or prevent subsequent settlement

File(s): Markets/Market/Market.sol

Description: The `_emergencyWithdraw` function drains all USDC to the owner and resets global pool accounting, but does not change `marketStatus` or set any flag to prevent the market from being settled afterward:

```
function _emergencyWithdraw() internal {
    if (block.timestamp < eventDate + EMERGENCY_TIMELOCK) {
        revert EmergencyTimelockNotMet();
    }

    uint256 balance = IERC20(usdcToken).balanceOf(address(this));
    IERC20(usdcToken).safeTransfer(owner(), balance);

    predictionsPoolBalance = 0;
    protocolFeesBalance = 0;
    curatorFeesBalance = 0;
    totalCostA = 0;
    totalCostB = 0;
    creditsUsedSideA = 0;
    creditsUsedSideB = 0;
    totalCreditsUsed = 0;

    emit EmergencyWithdrawal(owner(), balance);
}
```

After this function executes, the contract has zero USDC and zeroed global balances, but `marketStatus` remains unchanged (e.g., `CLOSED`), `settled` is still `false`, and all per-user state (`positions[].usdcSpent`, `balancesA/B`, `balanceCreditsAmount`) is preserved. This allows the multisig to subsequently call `_setWinner`, which sets `settled = true` and assigns an outcome.

Since `settled` remains `false`, the multisig can still call `_setWinner` afterward. If the market is settled with a winner, `predictionsPoolBalance` is zero, so all payouts compute to zero and revert with `NoPayoutAvailable`. The winner path has no credit restoration logic, so credits consumed by winning users are permanently lost.

If the market is settled as a draw, `withdraw()` computes `payout = totalUsdcSpent - creditsToBeReturned` from the preserved per-user state. This creates an asymmetry in how different user types are treated:

- Pure-credit users (where `payout == 0`) can successfully call `withdraw()`. The function restores their credits via `creditManager.addCredit(...)` and returns early before reaching the USDC transfer. These users are made whole;
- Mixed users (USDC + credits) and pure-USDC users have `payout > 0`, but the contract holds zero USDC. The transaction reverts with `InsufficientContractFunds`, and the revert rolls back any credit restoration that occurred earlier in the same call. These users lose both their USDC and their credits;

The result is that only pure-credit users can recover from an emergency-withdrawn market that is later settled as a draw, while all other users are locked out entirely. The credits restored to pure-credit users come at no cost to the owner, who already drained the contract's full USDC balance. The function also zeroes `totalCreditsUsed`, `creditsUsedSideA`, and `creditsUsedSideB` but does not interact with `CreditManager` to restore per-user credit balances. For non-pure-credit users, the only recovery path is for the `CreditManager` owner or multisig to manually call `addCredit` for each affected user using their `balanceCreditsAmount` values from the market contract.

Recommendation(s): Consider settling a market within `_emergencyWithdraw` to make the emergency state terminal and prevent subsequent settlement.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.14 [Low] LMSR math double-scales PRBMath exp() outputs, causing premature overflow in pricing and cost calculations

File(s): Markets/Market/libraries/MarketCalculations.sol

Description: getPrice() and getCostFunction() unwrap PRBMath exp() results into raw uint256 values (already scaled by 1e18) and then multiply by DECIMALS (1e18) again in intermediate steps:

```
function getPrice(...) public pure returns (uint256) {
    // ...
    uint256 expA = unwrap(udA.exp()); // already 1e18-scaled
    uint256 expB = unwrap(udB.exp());

    if (onSideA) {
        return (expA * DECIMALS) / (expA + expB); // expA * 1e18 overflows
    } else {
        return (expB * DECIMALS) / (expA + expB);
    }
}

function getCostFunction(...) external pure returns (uint256) {
    // ...
    uint256 expA = unwrap(udA.exp());
    uint256 expB = unwrap(udB.exp());

    UD60x18 udSum = ud((expA + expB) * DECIMALS); // (sum) * 1e18 overflows
    uint256 _logln = unwrap(udSum.ln());
    return (b * _logln);
}
```

The intermediate $\text{expA} * 1e18$ overflows uint256 when $\text{expA} > 1.15e59$, which occurs at $q/b > 94.5$. PRBMath's exp() supports inputs up to 133.08, so there is a window between 94.5 and 133.08 where exp() succeeds but the subsequent multiplication overflows and reverts. With a manual b of 1,000, this threshold is reached at roughly $q = 94,500$ (approximately \$94.5K invested on one side). Once triggered, getPrice(), getCostFunction(), and all functions that depend on them (predict(), sellPosition(), getSellReturn(), getOdds()) revert, freezing the market until an admin increases b.

In getPrice, the $* \text{DECIMALS}$ is necessary to preserve precision in the division but can be replaced with PRBMath's div() which uses mulDiv with 512-bit intermediate arithmetic. In getCostFunction, the $* \text{DECIMALS}$ is mathematically unnecessary — it adds a constant $\ln(1e18)$ to every evaluation — but is harmless in practice because LMSR only uses cost *differences* where the constant cancels. Removing it both fixes the overflow and corrects the formula.

Recommendation(s): In getPrice, replace the raw multiplication with PRBMath's overflow-safe div: `return unwrap(ud(expA)).div(ud(expA + expB))`. In getCostFunction, remove the extra scaling: `ud(expA + expB)` instead of `ud((expA + expB) * DECIMALS)`. Both changes produce identical pricing results and eliminate the overflow window entirely.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.15 [Low] No signer rotation or removal mechanism after one-time multisig configuration

File(s): Multisig/MultisigController.sol

Description: The configureMultisig(...) function enforces a one-time configuration per target through the targetConfigured flag:

```
function configureMultisig(
    address target,
    address[] memory _signers,
    uint256 _threshold,
    FunctionConfig[] memory _functionConfigs
) external override onlyOwner {
    // ...
    if (targetConfigured[target]) {
        revert TargetAlreadyConfigured();
    }
    // ...
    config.signers.push(_signers[i]);
    config.isSigner[_signers[i]] = true;
    // ...
    targetConfigured[target] = true;
}
```

Once configured, there is no function to add, remove, or replace signers, and no function to update the threshold. If a signer's private key is compromised, that key retains full signing authority for the lifetime of the MultisigController instance.

This means a compromised signer cannot be removed from the active signer set, and no target contract can be migrated to a new multisig configuration. A compromised signer retains permanent ability to contribute one valid signature toward any multisig operation.

Recommendation(s): Consider adding a multisig-governed signer rotation mechanism that allows the existing signer set to vote on adding or removing signers. Alternatively, consider adding a setMultisigController(...) function on target contracts (protected by the existing multisig) to allow migration to a new MultisigController instance with an updated signer set.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.16 [Low] Sell-generated protocol and curator fees are locked when a market resolves as a draw

File(s): Markets/Market/Market.sol

Description: During active trading, sells generate fees that are split: 50% to protocolFeesBalance and 50% recycled into predictionsPoolBalance:

```
uint256 protocolShare = fees / 2;
uint256 poolShare = fees - protocolShare;
protocolFeesBalance += protocolShare;
predictionsPoolBalance += poolShare;
```

When a market resolves as a draw, both _withdrawProtocolFees and withdrawCuratorFees revert:

```
function _withdrawProtocolFees() internal {
    if (!settled || isDraw) {
        revert EventNotFinishedOrDraw();
    }
    // ...
}
```

The comment on withdrawCuratorFees explains: "On draw, users get full refunds including fees, so curator cannot withdraw." This is correct for buy-side fees, users' usdcSpent includes the fees they paid on purchase, and draw refunds return usdcSpent. However, sell fees are collected from entirely separate transactions where the seller voluntarily exited their position and already received a reduced payout (after fees). These fees were earned from real completed trades and have no corresponding draw refund claimant.

The sell-fee portion of protocolFeesBalance remains locked in the contract with no withdrawal path. The only recovery mechanism is _emergencyWithdraw, which drains all funds indiscriminately and requires a 7-day timelock past the event date.

Recommendation(s): Consider tracking sell-generated protocol fees separately from buy-generated protocol fees. On draw, allow withdrawal of sell-generated fees only after verifying the contract's USDC balance still covers all outstanding draw obligations. This ensures the fees are only released when they are genuine surplus rather than needed to partially offset the draw-refund-overclaim deficit.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.17 [Low] Stepwise calculateDynamicB creates price discontinuities at tier boundaries

File(s): [Markets/Market/libraries/MarketCalculations.sol](#)

Description: The calculateDynamicB function determines the LMSR liquidity parameter b using a hardcoded lookup table of discrete tiers:

```
function calculateDynamicB(uint256 totalPoolSize) external pure returns (uint256) {
    if (totalPoolSize >= POOL_TIER_9) {
        return 1_000_000_000; // $100M+ -> b = 1B
    } else if (totalPoolSize >= POOL_TIER_8) {
        return 500_000_000; // $50M+ -> b = 500M
    } else if (totalPoolSize >= POOL_TIER_7) {
        return 100_000_000; // $10M+ -> b = 100M
    } // ... more tiers ...
    } else if (totalPoolSize >= POOL_TIER_1) {
        return 100_000; // $10K+ -> b = 100K
    } else if (totalPoolSize >= POOL_TIER_TINY) {
        return 10_000; // $1K+ -> b = 10K
    } else if (totalPoolSize >= POOL_TIER_MICRO) {
        return 1_000; // $100+ -> b = 1K
    } else {
        return 1_000; // below $100
    }
}
```

At boundary crossings, b jumps by up to 10x (e.g., 999 to 1,001 changes b from 1,000 to 10,000). Since b controls the curvature of the entire LMSR cost function, these discrete jumps break LMSR's fundamental property of continuous pricing. A pool at \$9,999 operates under $b = 10,000$, but a single \$2 trade that pushes it to \$10,001 switches the entire market to $b = 100,000$. Under the new b, prices flatten dramatically and the behavior of the market changes abruptly.

Users whose transactions land on different sides of a boundary experience fundamentally different pricing for similar positions. The abrupt 10x changes in market curvature at the most common pool sizes (\$1K and \$10K boundaries) make market behavior unpredictable.

Recommendation(s): Consider replacing the stepwise tier lookup with a continuous function for b calculations.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.18 [Low] Switching between manual and dynamic liquidity modes on a live market invalidates pool accounting

File(s): [Markets/Market/Market.sol](#)

Description: The multisig can toggle `_setAutoDynamicLiquidity(...)` at any time, including while the market is OPEN with active positions:

```
function _setAutoDynamicLiquidity(bool enabled) internal {
    autoDynamicLiquidity = enabled;
    emit AutoDynamicLiquidityChanged(enabled);
}
```

There is no restriction on market status. When a market uses a manually set liquidityParameter, gradual updates via `_setLiquidityParameter(...)` are expected to keep the market healthy as the pool grows. However, toggling from manual to dynamic mode (or vice versa) creates a discontinuity: the effective b jumps from the stored liquidityParameter to `calculateDynamicB(totalCostA + totalCostB)`, which can be an order-of-magnitude change if the pool has grown beyond the initial parameter's tier.

All existing accounting variables (`totalCostA`, `totalCostB`, `predictionsPoolBalance`) reflect LMSR costs accumulated under the previous b values. After a large discontinuity in effective b, subsequent sell operations compute state-dependent LMSR cost differences under the new b, which can diverge significantly from the tracked values. If a sell return exceeds the tracked `totalCostA`, the clamping logic (`else totalCostA = 0;`) silently absorbs the excess, leaving `totalCostA` at zero while shares remain outstanding.

Recommendation(s): Consider preventing switching between manual and dynamic liquidity modes on markets that have already received predictions. More broadly, consider whether the manual liquidity parameter system is necessary given that the dynamic system exists.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.19 [Low] Ternary market mint/merge operations hardcode all fees and price improvement to the YES position

File(s): [Orderbook/Exchange/OrderbookExchange.sol](#)

Description: In ternary (three-outcome) markets, fillOrders enforces a fixed token ID ordering where the taker must always be the YES (tokenId 0) position:

```
if (takerOrder.tokenId != 0 || makerOrder.tokenId != 1 || drawOrder.tokenId != 2) {
    revert InvalidTokenIds();
}
```

Because the YES order is always designated as the taker, all fee charges and price improvement logic are hardcoded to the YES position. The condition `takerOrder.tokenId == 0 || hasDrawOrder` is always true when `hasDrawOrder` is true, causing:

- Mint surplus is always deducted from `yesPay`, giving price improvement exclusively to the YES buyer;
- Merge surplus is always added to `yesProceeds` giving excess proceeds exclusively to the YES seller;
- Merge fees are always deducted from `yesProceeds`, charging the fee exclusively to the YES seller;
- Fee rates are always computed from the YES order's price via `_extractPrice(takerOrder)`;

In binary markets, the operator can alternate which order is designated as taker, distributing the fee burden fairly. In ternary markets, the protocol has no such flexibility due to the hardcoded token ID ordering.

Recommendation(s): Consider splitting fees proportionally across all three parties in ternary mint/merge operations based on each party's contribution, rather than concentrating the entire fee burden on the YES position. Alternatively, consider allowing the operator to specify which of the three orders is the taker rather than hardcoding the token ID ordering.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.20 [Low] sellPosition lacks slippage protection for sellers

File(s): [Markets/Market/Market.sol](#)

Description: The `predict` function provides buy-side slippage protection through the `maxPricePerShare` parameter. If the average cost per share exceeds the caller's limit, the transaction reverts:

```
function predict(
    bool onSideA,
    uint256 sharesToBuy,
    uint256 maxPricePerShare,
    PermitData calldata permit
) external override nonReentrant returns (uint256) {
    // ...
    if (maxPricePerShare > 0) {
        uint256 averagePrice = cost / sharesToBuy;
        if (averagePrice > maxPricePerShare) {
            revert SlippageExceeded(maxPricePerShare, averagePrice);
        }
    }
    // ...
}
```

The `sellPosition` function has no equivalent protection. It accepts only the side and share count, computes the return internally, and transfers whatever amount results:

```
function sellPosition(bool fromSideA, uint256 sharesToSell) external override nonReentrant returns (uint256) {
    // ...
    uint256 amountAfterFeesInternal = _processSell(fromSideA, sharesToSell);

    uint256 amountAfterFees = amountAfterFeesInternal / DECIMAL_MULTIPLIER;
    // ...
    IERC20(usdcToken).safeTransfer(msg.sender, amountAfterFees);

    return amountAfterFees;
}
```

A seller who previews their return via `getSellReturn()` sees a value computed from the market state at that moment. Between the preview and execution, other trades can change `sharesA`, `sharesB`, `totalCostA`, and `totalCostB`, which alter the LMSR cost curve and the effective `b` when `autoDynamicLiquidity` is enabled. The actual return at execution time can be lower than expected, and the seller has no on-chain mechanism to reject the trade.

Recommendation(s): Add a `minReturnAmount` parameter to `sellPosition` that reverts if the computed `amountAfterFees` falls below the caller's specified minimum.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.21 [Low] `setMultisigController` in `CreditManager` does not emit an event

File(s): `Credits/CreditManager.sol`

Description: The `setMultisigController` function sets the `multisigController` address, a critical security parameter that controls which address can execute privileged operations. However, it emits no event:

```
function setMultisigController(address _multisigController) external override onlyOwner {
    if (multisigController != address(0)) revert AlreadyInitialized();
    if (_multisigController == address(0)) revert ZeroAddress();
    multisigController = _multisigController;
}
```

Every other configuration function in `CreditManager` emits an event. The multisig controller configuration is the most privileged operation on the contract and is the only one invisible to event-based monitoring.

Recommendation(s): Consider defining and emitting a `MultisigControllerSet` event.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.22 [Info] 18-to-6 decimal truncation in `sellPosition` can burn shares for zero USDC

File(s): `Markets/Market/Market.sol`

Description: `sellPosition()` processes the sell and updates all state before converting the internal 18-decimal result to 6-decimal USDC via floor division. `_processSell()` reduces `sharesA/B`, `balancesA/B`, `totalCostA/B`, and `predictionsPoolBalance`, then the result is floored:

```
uint256 amountAfterFeesInternal = _processSell(fromSideA, sharesToSell);
uint256 amountAfterFees = amountAfterFeesInternal / DECIMAL_MULTIPLIER;
// ...
IERC20(usdcToken).safeTransfer(msg.sender, amountAfterFees);
```

If `amountAfterFeesInternal < 1e12`, the user's shares are burned and pool accounting is updated, but the transfer sends 0 USDC. There is no zero-check, the transaction succeeds silently.

This is reachable when the liquidity parameter `b` is very large relative to the position size. With `b = MAX_LIQUIDITY_PARAMETER (1e9)`, selling 1 share returns approximately `1e9` internally — well below the `1e12` threshold. A more natural path: a user buys a small position while `b` is low, then dynamic `b` scales up as the pool grows, reducing the per-share cost difference under the new `b`.

Recommendation(s): Consider adding a check such as `if (amountAfterFees == 0) revert ZeroPayout();` after the 18-to-6 conversion, before the transfer.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.23 [Info] Curator fees are not subject to token-based fee discounts

File(s): Markets/Market/Market.sol

Description: In `_getPredictionCosts`, the protocol fee applies the user's discount from governance token holdings, while the curator fee does not:

```
if (feesEnabled) {
    uint256 discountBps = getFeeDiscountBasisPoints(msg.sender);
    _protocolFee = (cost * protocolFee * (10000 - discountBps)) / 1e8;

    if (curator != address(0) && curatorFee > 0) {
        curatorFeeAmount = (cost * curatorFee) / 10000;
    }
}
```

The curator fee is calculated without any discount regardless of the user's token holdings. This means users holding governance/discount tokens receive reduced protocol fees but always pay the full curator fee. This is an asymmetry that could be unexpected to users who assume "fee discount" applies to total fees.

Recommendation(s): Consider documenting this behavior clearly in the UI and contract documentation. If curator fee discounts are desired, apply the same `discountBps` logic to `curatorFeeAmount`.

Status: Acknowledged

Update from the client: By design.

6.0.24 [Info] EIP-2612 permit flow in `predict(...)` is broken for users with credit balances

File(s): Markets/Market/Market.sol

Description: The `predict(...)` function supports EIP-2612 permits, allowing users to approve and transfer USDC in a single transaction. However, the permit is called with `usdcAmountRequired` the post-credit-deduction amount which the user cannot know at signing time.

The credit deduction happens internally inside `predict(...)` after the user has already signed the permit:

```
function predict(
    bool onSideA,
    uint256 sharesToBuy,
    uint256 maxPricePerShare,
    PermitData calldata permit
) external override nonReentrant returns (uint256) {
    // ...
    (
        uint256 cost,
        uint256 actualProtocolFee,
        uint256 totalAmountRequiredInUSDC,
        uint256 curatorFeeAmount
    ) = calculateAmountRequiredInUSDC(sharesToBuy, onSideA);
    // ...
    uint256 creditsUsed = 0;
    uint256 userCredits = creditManager.getCreditBalance(msg.sender);

    if (userCredits > 0) {
        creditsUsed = Math.min(totalAmountRequiredInUSDC, userCredits);
    }

    uint256 usdcAmountRequired = totalAmountRequiredInUSDC - creditsUsed;
    // ...
    if (usdcAmountRequired > 0) {
        if (permit.deadline > 0) {
            IERC20Permit(usdcToken).permit(
                msg.sender,
                address(this),
                usdcAmountRequired,
                permit.deadline,
                permit.v,
                permit.r,
                permit.s
            );
        }

        IERC20(usdcToken).safeTransferFrom(msg.sender, address(this), usdcAmountRequired);
    }
    // ...
}
```

The user's only available information before calling `predict(...)` is `totalAmountRequiredInUSDC` returned by `calculateAmountRequiredInUSDC(...)`. They sign a permit for that full amount. Inside `predict(...)`, the contract deducts credits and calls `permit()` with the reduced `usdcAmountRequired`. Because the signed value does not match the value passed to `permit()`, the call reverts with `ERC2612InvalidSigner`.

The user's credit balance can also change between signing and execution for example, if another market consumes credits in an earlier transaction within the same block, or if the protocol adds credits between the user signing and the transaction being mined making the post-credit amount fundamentally unpredictable at signing time.

As a result, any user with a non-zero credit balance is forced to fall back to a separate `approve()` transaction with `permit.deadline = 0`, which costs extra gas and defeats the purpose of the permit flow.

Recommendation(s): Consider signing and calling the permit for the full `totalAmountRequiredInUSDC` amount, or providing a way for users to query the exact post-credit USDC amount before signing.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.25 [Info] Fee discount can be obtained via flash-loaned tokens

File(s): [Markets/Market/Market.sol](#), [Markets/Market/libraries/MarketCalculations.sol](#)

Description: The fee discount mechanism checks a user's token balance at the moment of the call via `IERC20(token).balanceOf(user)`:

```
function getFeeDiscountBasisPoints(
    address user,
    mapping(address => TokenDiscountConfig) storage tokenConfigs,
    address[] memory discountTokens
) external view returns (uint256) {
    // ...
    for (uint256 i = 0; i < discountTokens.length; i++) {
        // ...
        try IERC20(token).balanceOf(user) returns (uint256 balance) {
            tokenBalance = balance;
        } catch {
            continue;
        }
        // ...
    }
    return maxDiscount;
}
```

This balance check is a point-in-time snapshot. A user can borrow discount tokens via a flash loan, call `predict(...)` or `sellPosition(...)` through an intermediary contract to receive the reduced fee, and return the tokens in the same transaction.

The discount applies to both buy-side protocol fees (in `calculateAmountRequiredInUSDC`) and sell-side dynamic fees (in `_calculateSellFees`), meaning both entry and exit costs can be reduced this way.

This allows any user to obtain the maximum configured fee discount without actually holding the required tokens, undermining the intended incentive for long-term token holders.

Recommendation(s): Consider implementing a time-weighted average balance check, a snapshot-based mechanism, or requiring that discount tokens be staked/locked for a minimum period before the discount applies.

Status: Acknowledged

Update from the client: We consider the practical risk negligible

6.0.26 [Info] Hardcoded magic numbers used throughout the codebase

File(s): [Markets/Market/Market.sol](#), [Markets/Market/libraries/MarketCalculations.sol](#), [Multisig/MultisigController.sol](#), [Orderbook/Exchange/OrderbookExchange.sol](#), [Orderbook/Market/OrderbookMarket.sol](#), [Orderbook/Factory/OrderbookFactory.sol](#), [Credits/CreditManager.sol](#)

Description: Multiple contracts across the codebase use hardcoded numeric literals directly in logic rather than referencing named constants. The most common categories are:

- Basis point denominators (e.g., 10000) and fee/ratio thresholds (e.g., 2000, 3000, 5000) used in conditionals and arithmetic in [Market.sol](#), [MarketCalculations.sol](#), [OrderbookMarket.sol](#), and [OrderbookFactory.sol](#);
- Time durations (e.g., 3 days, 7 days, 4 days) used inline for sell lockouts, holding period caps, and fee multiplier brackets in [Market.sol](#) and [MarketCalculations.sol](#);
- Fee multiplier values (e.g., 50, 75, 100, 125, 150, 200, 300) for dynamic sell fee calculation in [MarketCalculations.sol](#);
- Function selectors written as raw hex literals (e.g., 0x1aa48bde, 0x1956e2ae) for multisig dispatch routing in [Market.sol](#), [MultisigController.sol](#), [OrderbookExchange.sol](#), [OrderbookMarket.sol](#), and [CreditManager.sol](#);
- Calculation divisors (e.g., 1e8, 1e14, 1e18) embedded directly in fee and price formulas in [Market.sol](#), [MarketCalculations.sol](#), and [OrderbookExchange.sol](#);
- Cancellation refund splits (e.g., 3333, 3334, 5000) used for ternary outcome distributions in [OrderbookMarket.sol](#);

Using named constants improves readability, reduces the risk of inconsistent updates, and makes the intended semantics of each value explicit.

Recommendation(s): Consider extracting all hardcoded numeric literals into named constants with descriptive names (e.g., BASIS_POINTS, MAX_SELL_FEE_BPS, SELL_LOCKOUT_BEFORE_EVENT, HOLDING_MULTIPLIER_7D). For function selectors, consider computing them from the function signature string (e.g., bytes4(keccak256("setWinner(bool,bool)"))) or referencing interface definitions to prevent silent breakage if function signatures change.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.27 [Info] Implementation contracts can be initialized by anyone

File(s): [Markets/Market/Market.sol](#), [Orderbook/Market/OrderbookMarket.sol](#), [Orderbook/Exchange/OrderbookExchange.sol](#)

Description: The protocol deploys implementation contracts as templates for EIP-1167 clones. These implementation contracts use empty constructors and an initialized boolean to guard their initialize function:

```
// Market.sol
constructor() Ownable(msg.sender) {}

function initialize(...) external override {
    if (initialized) {
        revert AlreadyInitialized();
    }
    // ...
}
```

```
// OrderbookMarket.sol
constructor() {}

function initialize(...) external override {
    if (initialized) revert AlreadyInitialized();
    // ...
}
```

```
// OrderbookExchange.sol
constructor() EIP712("PalpitadaOrderbookExchange", "1") {}

function initialize(...) external {
    if (initialized) revert AlreadyInitialized();
    // ...
}
```

None of these constructors set `initialized = true`, so `initialize()` remains callable on the implementation contract itself. Since `initialized` defaults to `false`, anyone can call `initialize()` on each implementation with arbitrary parameters, setting themselves as owner, configuring a controlled multisig, and pointing to arbitrary token addresses.

This does not affect clones, as each clone has independent storage. However, the implementation addresses are publicly visible on-chain and could be used in social engineering scenarios where a user is tricked into interacting with what appears to be a legitimate contract at a known protocol address.

Recommendation(s): Consider setting `initialized = true` in each implementation's constructor.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.28 [Info] Inconsistent `msg.sender` vs `_msgSender()` usage in `CreditManager`'s `onlyAuthorized` modifier

File(s): [Credits/CreditManager.sol](#)

Description: The `onlyAuthorized` modifier mixes `msg.sender` and `_msgSender()` in the same conditional:

```
modifier onlyAuthorized() {
    if (!isAuthorizedConsumer[msg.sender] && owner() != _msgSender()) {
        revert NotAuthorized();
    }
    _;
}
```

The `onlyMarketFactory` modifier consistently uses `_msgSender()`:

```
modifier onlyMarketFactory() {
    if (_msgSender() != marketFactory) {
        revert NotAuthorized();
    }
    _;
}
```

In the current deployment, `msg.sender` and `_msgSender()` return the same value because no meta-transaction forwarder is configured. However, if `ERC2771Context` or a similar forwarder is added in the future, `msg.sender` (the forwarder) and `_msgSender()` (the original signer) would diverge, causing the authorized consumer check to use the wrong address.

Recommendation(s): Consider using `_msgSender()` consistently in both conditions within the `onlyAuthorized` modifier to avoid inconsistent behavior if meta-transaction support is introduced.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.29 [Info] Inconsistent validation for `protocolFee` between `initialize(...)` and `_setProtocolFee(...)`

File(s): [Markets/Market/Market.sol](#)

Description: During initialization, `protocolFee` is allowed to be zero:

```
function initialize(...) external override {
    // ...
    if (_protocolFee >= 10000) {
        revert InvalidFees();
    }
    // ...
}
```

However, `_setProtocolFee(...)` (called via multisig to update the fee after deployment) explicitly rejects zero:

```
function _setProtocolFee(uint256 _protocolFee) internal {
    if (_protocolFee == 0) {
        revert ZeroValue();
    }
    if (_protocolFee >= 10000) {
        revert InvalidFees();
    }
    // ...
}
```

A market initialized with `protocolFee = 0` cannot be restored to that state through `_setProtocolFee(...)`. Conversely, if the intent is that zero fees should never be allowed, the check is missing from `initialize(...)`.

Recommendation(s): Consider aligning the validation between both functions.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.30 [Info] Market hard-codes a 6-decimal token assumption via DECIMAL_MULTIPLIER

File(s): [Markets/Market/Market.sol](#), [Markets/Market/MarketStorage.sol](#)

Description: The contract uses a fixed `DECIMAL_MULTIPLIER = 1e12` to convert between 18-decimal internal accounting and the payment token's decimals, assuming the token always has 6 decimals. This constant is used throughout the protocol for charging users, computing payouts, and converting pool balances:

```
uint256 public constant DECIMAL_MULTIPLIER = 1e12;

// Charging (round up):
uint256 totalAmountRequiredInUSDC = (totalAmountRequired + DECIMAL_MULTIPLIER - 1) / DECIMAL_MULTIPLIER;

// Paying out (truncate):
uint256 amountAfterFees = amountAfterFeesInternal / DECIMAL_MULTIPLIER;
```

If a token with different decimals is configured, internal accounting and actual token transfers would diverge silently. Since `initialize()` accepts any `_usdcToken` address without validating its decimals, there is no on-chain safeguard against misconfiguration.

Recommendation(s): If the protocol plans to support payment tokens beyond 6-decimal USDC, consider accepting the token's decimal count as an initialization parameter and deriving the conversion factor dynamically rather than relying on a hard-coded constant. Note that `decimals()` is optional in the ERC-20 standard and not all tokens implement it, so querying it on-chain may not be reliable and an explicit parameter validated by the deployer is a safer approach.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.31 [Info] MarketFactory owner storage variable has no setter preventing rotation for market ownership

File(s): [Markets/Factory/MarketFactory.sol](#), [Markets/Factory/MarketFactoryStorage.sol](#),

Description: `MarketFactoryStorage` declares a plain address `public owner` variable that is assigned once in the constructor and passed to every cloned `Market`'s `initialize(...)` as its `Ownable` owner:

```
// MarketFactoryStorage.sol
address public owner;

// MarketFactory.sol constructor
owner = msg.sender;

// MarketFactory.sol createMarket
Market(newMarket).initialize(
    // ...
    owner,
    // ...
);
```

In `Market.sol`, this address receives `Ownable` ownership via `_transferOwnership(owner)`.

`MarketFactory` uses `AccessControl` for role management. However, the `owner` storage variable has no setter function. After an admin rotation, all newly created markets still receive the original deployer as their `Ownable` owner. The new admin has no way to set itself as the owner of future markets without redeploying the entire factory.

While each `Market`'s ownership can be changed post-deployment via `Ownable`'s `transferOwnership(...)`, this requires the stale owner to cooperate and must be done individually per market.

Recommendation(s): Consider adding a setter function for `owner` so that owner rotation is reflected in future market deployments.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.32 [Info] No combined ceiling on protocolFee + curatorFee

File(s): [Markets/Market/Market.sol](#), [Markets/Factory/MarketFactory.sol](#)

Description: Market.initialize validates protocolFee independently (`_protocolFee >= 10000` reverts) but does not check the sum of protocolFee + curatorFee. Both fees are in basis points, and each is capped under 10000 individually, so their combined value could reach up to 19998 bps (200% of the cost).

In `_getPredictionCosts`:

```
uint256 totalAmountRequired = cost + _protocolFee + curatorFeeAmount;
```

If protocolFee = 9999 (99.99%) and curatorFee = 9999 (99.99%), the total amount required would be approximately three times the LMSR cost. While such extreme fees would be set by the market creator and visible to users before they participate, the protocol has no guardrail preventing accidental or malicious over-fee configuration.

Recommendation(s): Consider adding a combined fee ceiling check (e.g., `protocolFee + curatorFee < MAX_COMBINED_FEE`) in Market.initialize to prevent markets from being created with unreasonable total fee rates.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.33 [Info] No on-chain order cancellation for regular orders

File(s): [Orderbook/Exchange/OrderbookExchange.sol](#)

Description: Regular Order structs have no on-chain cancellation mechanism. The nonce field is part of the signed hash but is never validated against an on-chain counter. Orders with `expiration = 0` exist forever. The only way to cancel is off-chain by asking the operator to stop matching. If the operator is compromised, stale orders can be filled at outdated prices and users have no way to revoke signed orders on-chain.

Recommendation(s): Consider adding an `invalidateNonce(uint256 nonce)` function that lets users increment their nonce, invalidating all orders with lower nonces.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.34 [Info] Non-delayed operations invalidate pending delayed operations via shared nonce

File(s): [Multisig/MultisigController.sol](#)

Description: The MultisigController uses a single nonce counter per target for all operations. When a delayed operation is queued, the nonce is not incremented until the operation actually executes:

```
function executeWithMultisig(Operation memory operation, bytes[] memory signatures)
    external
    override
    returns (bool)
{
    _validateOperation(operation);
    // ...
    if (!_checkAndHandleDelay(operation.target, operation.selector, operationHash)) {
        return false; // Operation queued, not executed -- nonce NOT incremented
    }

    executedOperations[operation.target][operationHash] = true;
    nonces[operation.target]++;
    // ...
}
```

While a delayed operation sits in the queue waiting for its delay to elapse, the target nonce remains unchanged. If during this window the signers execute any other non-delayed operation for the same target, the nonce advances and the pending delayed operation becomes permanently unexecutable:

1. Signers queue `setWinner(...)` with a 1-day delay (nonce = 5). The operation is queued but the nonce stays at 5;
2. During the delay, signers execute `setSellsEnabled(true)` (non-delayed, nonce = 5). This succeeds and increments the nonce to 6;
3. After the delay elapses, the queued `setWinner` operation (nonce = 5) fails validation because `5 != nonces[target]` (now 6);
4. The `setWinner` must be re-signed with nonce 6 and re-queued, restarting the full delay period;

The stale pendingOperations and operationQueued entries for the invalidated operation also remain in storage and are never cleaned up.

Recommendation(s): Consider documenting this behavior clearly for operators so they understand that executing any operation during a pending delay period will invalidate the queued operation.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.35 [Info] OrderbookMarket emergency withdrawal has no timelock

File(s): [Orderbook/Market/OrderbookMarket.sol](#)

Description: In Market.sol (LMSR markets), _emergencyWithdraw enforces a 7-day timelock after the event date to give time for normal settlement before funds can be drained:

```
function _emergencyWithdraw() internal {
    if (block.timestamp < eventDate + EMERGENCY_TIMELOCK) {
        revert EmergencyTimeLockNotMet();
    }
    // ...
}
```

The equivalent function in OrderbookMarket has no such restriction:

```
function _emergencyWithdraw(address recipient) internal {
    if (recipient == address(0)) revert ZeroAddress();

    uint256 balance = IERC20(usdc).balanceOf(address(this));
    if (balance == 0) revert ZeroAmount();

    totalUsdcReserve = 0;
    IERC20(usdc).safeTransfer(recipient, balance);

    emit EmergencyWithdrawal(recipient, balance);
}
```

The multisig can execute an emergency withdrawal at any time, including while the market is active and users are holding positions. While MultisigController may impose a time delay on the operation itself, this is a configurable per-function delay set during deployment and is not tied to the market's lifecycle. A market configured with a short or zero delay for emergencyWithdraw would allow immediate fund extraction.

Recommendation(s): Consider adding a timelock tied to the market's lifecycle.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.36 [Info] The executeMultisigOperation(...) function bypasses initialization checks for marketFactory

File(s): Credits/CreditManager.sol

Description: The CreditManager contract is designed to have a one-time initialization for the marketFactory address. The setMarketFactory(...) function, which is restricted to the contract owner, explicitly prevents re-initialization by checking if marketFactory is already set.

```
function setMarketFactory(address _marketFactory) external override onlyOwner {
    // @audit-issue This check prevents re-initialization by the owner.
    if (marketFactory != address(0)) revert AlreadyInitialized();
    _setMarketFactory(_marketFactory);
}
```

However, the executeMultisigOperation(...) function allows a multisigController to call the internal _setMarketFactory(...) function directly without performing this "already initialized" check.

```
function executeMultisigOperation(bytes4 selector, bytes calldata data)
    external override returns (bool)
{
    if (msg.sender != multisigController) revert NotAuthorized();

    if (selector == 0x4dcc434a) { // setMarketFactory(address)
        address _marketFactory = abi.decode(data, (address));
        // @audit-issue This internal call bypasses the alreadyInitialized check.
        _setMarketFactory(_marketFactory);
    }
    // ...
}
```

If the intention of the protocol is to ensure that the marketFactory is an immutable, one-time set parameter for security reasons, the current implementation allows the multisig to bypass this constraint and change the factory at any time. If this bypass is an intended administrative "backdoor" for the multisig, it should be explicitly documented to avoid confusion regarding the contract's initialization state.

Recommendation(s): Consider adding the if (marketFactory != address(0)) revert AlreadyInitialized(); check inside the executeMultisigOperation(...) block for the 0x4dcc434a selector if the parameter is intended to be strictly one-time. Alternatively, if the multisig is supposed to have the power to rotate the factory, update the documentation to reflect that the initialization check only applies to the standard Ownable caller and not the multisig.

Status: Acknowledged

Update from the client: By design.

6.0.37 [Info] MAX_SHARES overflow guard is not compatible with the allowed liquidityParameter range

File(s):

Description: The contract uses a fixed MAX_SHARES limit (13,000,000,000) as an overflow guard, but the actual overflow condition in the LMSR math depends on the ratio shares / b used as the input to UD60x18.exp().

```
// MarketStorage.sol
uint256 public constant MIN_LIQUIDITY_PARAMETER = 10;
uint256 public constant MAX_SHARES = 13_000_000_000;
```

Pricing and cost compute exponentials on (shares * 1e18) / b:

```
// MarketCalculations.sol
UD60x18 udA = ud((qA * DECIMALS) / b);
UD60x18 udB = ud((qB * DECIMALS) / b);
uint256 expA = unwrap(udA.exp());
uint256 expB = unwrap(udB.exp());
```

UD60x18.exp() reverts when its input exceeds its maximum supported value. For small b values (the contract allows b as low as MIN_LIQUIDITY_PARAMETER = 10), even relatively small sharesA/sharesB values can push (shares * 1e18) / b above the exp() limit, causing getPrice, getCostFunction, predict(), sellPosition(), and getSellReturn() to revert.

Because _setLiquidityParameter(uint256 newB) permits setting b down to 10:

```
function _setLiquidityParameter(uint256 newB) internal {
    if (newB < MIN_LIQUIDITY_PARAMETER || newB > MAX_LIQUIDITY_PARAMETER) {
        revert InvalidLiquidityParameter();
    }
}
```

```
}  
    liquidityParameter = newB;  
}
```

a misconfiguration (or a later reduction of `b` after shares have grown) can effectively brick the market's pricing/trading functions due to math reverts, despite `MAX_SHARES` checks still passing.

Recommendation(s): Consider making `MAX_SHARES` dynamic based on the current `b` value, or adding a validation that checks the existing share counts against the new `b` to ensure calculation remains within `exp()`'s safe input range.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.38 [Info] WinnerSet event is misleading when a draw is declared

File(s): [Markets/Market/Market.sol](#)

Description: The `_setWinner(...)` function emits a single `WinnerSet` event for all settlement outcomes, including draws:

```
function _setWinner(bool sideAHasWon_, bool draw) internal {  
    // ...  
    if (draw) {  
        isDraw = true;  
    } else {  
        sideAHasWon_ = sideAHasWon_;  
    }  
    settled = true;  
    marketStatus = MarketStatus.SETTLED;  
    emit MarketStatusChanged(MarketStatus.CLOSED, MarketStatus.SETTLED);  
    emit WinnerSet(sideAHasWon_, draw);  
}
```

When `draw` is true, the `sideAHasWon_` parameter is not used in state updates, but it is still included in the emitted event. This means a draw can produce `WinnerSet(true, true)` or `WinnerSet(false, true)` depending on the caller's input, both representing the same outcome but appearing different to off-chain consumers indexing the event. The event name itself is also semantically incorrect for a draw, as there is no winner.

By comparison, the Orderbook market emits a `Resolved(WinningOutcome, uint256)` event that clearly distinguishes all three outcomes through the enum value.

Recommendation(s): Consider emitting a separate `DrawDeclared` event when a draw is settled, or renaming the event to something like `MarketResolved` and normalizing the `sideAHasWon_` parameter to a consistent value (e.g., `false`) when draw is true.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.39 [Info] isRedeemOrderValid returns false for cancelled markets despite executeRedeem succeeding

File(s): Orderbook/Exchange/OrderbookExchange.sol, Orderbook/Market/OrderbookMarket.sol

Description: The isRedeemOrderValid view function only checks whether a market is resolved via isResolved():

```
try IOrderbookMarket(redeemOrder.market).isResolved() returns (bool resolved) {
    if (!resolved) return false;
} catch {
    return false;
}
```

However, redeemFor(...) in OrderbookMarket successfully processes redemptions for both resolved and cancelled markets:

```
bool isResolvedMarket = resolution.resolved;
bool isCancelledMarket = status == OrderbookMarketStatus.CANCELLED;

if (!isResolvedMarket && !isCancelledMarket) {
    revert MarketNotResolved(); //..
```

When a market is cancelled via _cancel(), only status is set to CANCELLED, resolution.resolved remains false. So isResolved() returns false and isRedeemOrderValid returns false, even though executeRedeem would succeed.

Since redeemFor is onlyExchange, users cannot redeem directly. If the backend relies on isRedeemOrderValid as a gate, it will skip all cancelled-market redemptions, preventing users from claiming their refunds through the normal gasless flow.

Recommendation(s): Consider updating isRedeemOrderValid to also check for the cancelled state, for example by adding an isCancelled() check alongside isResolved().

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.40 [Info] minimumOrderAmount stored but never enforced

File(s): Orderbook/Exchange/OrderbookExchange.sol

Description: minimumOrderAmount is initialized in initialize() and settable via multisig (_setMinimumOrder), but never checked in fillOrders() or any execution path. It is a dead state variable — the setter and storage exist but have no effect on protocol behavior.

Recommendation(s): Either enforce the minimum in fillOrders() or remove the variable entirely.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.41 [Best Practices] Enum types lack a default sentinel value for uninitialized state

File(s): Orderbook/Market/interfaces/IOrderbookMarketStructs.sol, Orderbook/Exchange/interfaces/IOrderbookExchangeEnums.sol, Orderbook/Market/interfaces/IOrderbookMarketStructs.sol, Markets/Market/interfaces/IMarketEnums.sol

Description: Several enum definitions across the codebase assign meaningful values at index 0, which is also the default value for any uninitialized enum in Solidity storage. This makes it impossible to distinguish between an explicitly set value and one that was never assigned.

As an example, WinningOutcome:

```
enum WinningOutcome {
    SIDE_A, // 0 - YES wins (binary) or HOME wins (ternary)
    SIDE_B, // 1 - NO wins (binary) or AWAY wins (ternary)
    DRAW    // 2 - Only valid for ternary markets
}
```

An uninitialized WinningOutcome variable silently reads as SIDE_A, which represents a real resolution. Any code path that reads this value before the market is resolved would interpret it as "Side A wins" rather than "no winner has been set."

Other enums follow the same pattern: MarketStatus defaults to OPEN, Side defaults to BUY, MatchType defaults to DIRECT, and MarketType defaults to LMSR. While MarketStatus defaulting to OPEN happens to align with the intended lifecycle of a newly deployed market, the remaining enums carry real semantic meaning at index 0 with no way to represent an unset state.

Recommendation(s): Consider prepending a NONE or UNSET member at index 0 for enums where an uninitialized default could be confused with a valid state.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.42 [Best Practices] Fee configuration allows setting topicRate to zero after registration

File(s): [Orderbook/Exchange/OrderbookExchange.sol](#)

Description: When registering a market, `_registerMarket` requires a non-zero `topicRate`:

```
function _registerMarket(
    address market,
    uint256 topicRate,
    bool dynamicPricingEnabled
) internal {
    if (market == address(0)) revert ZeroAddress();
    if (registeredMarkets[market]) revert MarketAlreadyRegistered();
    if (topicRate == 0) revert MarketFeeConfigRequired();
    // ...
}
```

However, `_setMarketFeeConfig` (callable via multisig) does not enforce the same constraint:

```
function _setMarketFeeConfig(
    address marketAddr,
    uint256 topicRate,
    bool dynamicEnabled
) internal {
    if (!registeredMarkets[marketAddr]) revert MarketNotRegistered();

    marketFeeConfigs[marketAddr] = MarketFeeConfig({ topicRate: topicRate, dynamicPricingEnabled: dynamicEnabled });
    emit MarketFeeConfigSet(marketAddr, topicRate, dynamicEnabled);
}
```

The multisig can set `topicRate = 0` after registration, which disables all fees for that market. This is inconsistent with the registration-time validation.

Recommendation(s): Consider aligning the validation between both functions.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.43 [Best Practices] No mechanism to deauthorize markets from CreditManager

File(s): [Credits/CreditManager.sol](#), [Markets/Factory/MarketFactory.sol](#)

Description: When `MarketFactory` creates a market, it authorizes it as a credit consumer in `CreditManager`:

```
CreditManager creditManager = CreditManager(CREDIT_MANAGER);
creditManager.setAuthorizedConsumer(newMarket);
```

The `setAuthorizedConsumer` function can only set authorization to true:

```
function setAuthorizedConsumer(address _consumer) external override onlyMarketFactory nonReentrant {
    if (_consumer == address(0)) revert ZeroAddress();
    isAuthorizedConsumer[_consumer] = true;
    emit NewConsumer(_consumer);
}
```

There is no `removeAuthorizedConsumer` function, and the `executeMultisigOperation` dispatcher only handles the `setMarketFactory`, `addCredit`, and `consumeCredit` selectors. Once a market is authorized, it retains the ability to call `addCredit` and `consumeCredit` indefinitely.

Authorized markets can both consume user credits (reducing balances) and add credits (inflating balances). If a market needs to be decommissioned, its credit authorization persists with no revocation path. The only recovery would be deploying an entirely new `CreditManager`, losing all existing credit balances and requiring re-authorization of every legitimate market.

Recommendation(s): Consider adding a `removeAuthorizedConsumer(address)` function gated by the multisig controller.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.44 [Best Practice] Non-standard double-hash EIP-712

File(s): Multisig/MultisigController.sol

Description: MultisigController wraps the EIP-712 structured hash inside an additional 19Ethereum Signed Message:32 hash. This creates a non-standard signature scheme that may confuse wallet integrations expecting pure EIP-712. The Orderbook system correctly uses standard EIP-712 via OpenZeppelin's EIP712 base contract. Wallets or signing libraries that implement EIP-712 natively will not produce the correct signature format without additional wrapping logic.

Recommendation(s): Consider aligning MultisigController with the Orderbook's approach and use OpenZeppelin's EIP712 base contract for standard-compliant structured hashing.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.45 [Best Practices] OrderbookFactory allows 100% protocol fee at market creation

File(s): Orderbook/Factory/OrderbookFactory.sol, Markets/Market/Market.sol

Description: The OrderbookFactory validates the protocol fee with an inclusive upper bound:

```
// OrderbookFactory.sol
if (protocolFee > 10000 || protocolFee == 0 || topicRate == 0) revert InvalidProtocolFee();
```

This allows protocolFee = 10000 (100%), meaning the operator could charge 100% of trade value as fees. In contrast, the AMM Market contract uses a strict upper bound:

```
// Market.sol initialize
if (_protocolFee >= 10000) {
    revert InvalidFees();
}
```

This inconsistency means AMM markets enforce a strictly-less-than-100% fee, while orderbook markets allow exactly 100%.

Recommendation(s): Consider aligning the validation to use a strict upper bound consistently across both factories, or document the intentional difference.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.46 [Best Practices] OrderbookMarket transferFrom silently no-ops for invalid token IDs

File(s): Orderbook/Market/OrderbookMarket.sol

Description: transferFrom handles token IDs 0 (YES), 1 (NO), and 2 (DRAW), but has no default revert for unrecognized values:

```
function transferFrom(
    address from,
    address to,
    uint256 amount,
    uint8 tokenId
) external override onlyExchange whenActive whenNotResolved whenNotExpired {
    if (tokenId == 0) {
        // YES token
        // ...
    } else if (tokenId == 1) {
        // NO token
        // ...
    } else if (tokenId == 2) {
        // DRAW token (only for ternary markets)
        if (!hasDraw) revert DrawNotAllowed();
        // ...
    }

    emit Transfer(from, to, amount, tokenId);
}
```

If tokenId >= 3, none of the branches execute. The function emits a Transfer event and returns without moving any tokens. In a DIRECT trade, the buyer's USDC is transferred to the seller before transferFrom is called. If the token transfer silently fails, the buyer pays USDC but receives no tokens.

This requires a user to sign an order with an invalid tokenId, so it is a self-harm scenario rather than an external exploit. However, the function should fail explicitly rather than succeed silently.

Recommendation(s): Add a revert for unrecognized token IDs after the last `else if` branch.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.47 [Best Practices] State changes in the `predict(...)` function are ineffective due to an immediate revert

File(s): [Markets/Market/Market.sol](#)

Description: The `predict(...)` function is the primary entry point for users to participate in a market. It includes a check to prevent predictions from being placed after the `eventDate` has passed.

```
function predict(...) external override nonReentrant returns (uint256) {
    // ...
    if (block.timestamp > eventDate) {
        // Auto-close market if past event date
        // @audit-issue These state changes and events are reverted by the following line.
        marketStatus = MarketStatus.CLOSED;
        emit MarketStatusChanged(MarketStatus.OPEN, MarketStatus.CLOSED);
        revert EventClosed();
    }
    // ...
}
```

As shown in the code snippet above, when the current time exceeds the `eventDate`, the contract attempts to update the `marketStatus` to `MarketStatus.CLOSED` and emit a `MarketStatusChanged` event. However, the function immediately executes `revert EventClosed()`. In the EVM, a revert undoes all state modifications made during the execution of a transaction. Consequently, the assignment to `marketStatus` is rolled back, and the event is never stored in the blockchain logs.

This results in the "auto-close" logic being a no-op. The market status will remain in its previous state until a successful transaction, such as an administrator calling `_setWinner(...)` or `_closeMarket(...)`, transitions the status.

Recommendation(s): Consider removing the state update and event emission from within the reverting `if` block to save gas and improve code clarity.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.48 [Best Practices] The `depositUSDC(...)` function is redundant

File(s): [Markets/Market/Market.sol](#)

Description: The Market contract provides the `depositUSDC(...)` function, which is restricted to the contract owner, to allow the manual transfer of USDC into the contract's balance.

```
function depositUSDC(uint256 amount) external override onlyOwner nonReentrant {
    IERC20(usdcToken).safeTransferFrom(msg.sender, address(this), amount);
    emit USDCDeposited(msg.sender, amount);
}
```

However, the function only executes a `safeTransferFrom(...)` call and emits an event. It does not update any internal accounting variables. Since the contract can receive USDC through standard token transfers, this function is functionally redundant.

This results in unnecessary gas consumption for the owner due to the `onlyOwner` and `nonReentrant` modifiers.

Recommendation(s): Consider removing the `depositUSDC(...)` function to simplify the codebase and reduce gas costs.

Status: Acknowledged

Update from the client: We acknowledge the function is technically redundant — USDC can be transferred directly to the contract. However, we will keep it for operational reasons.

6.0.49 [Best Practice] Wrong error emitted in `withdrawCuratorFees`

File(s): [Markets/Market/Market.sol](#)

Description: The check `msg.sender != curator` emits `NotOwner()` error instead of a more appropriate `NotCurator()` error. This causes misleading revert messages for off-chain tooling and front-ends attempting to display why a withdrawal failed.

Recommendation(s): Consider defining and emitting a `NotCurator()` custom error in place of `NotOwner()`.

Status: Fixed

Update from the client: Fixed in [PR](#)

6.0.50 [Best Practices] `_cancel()` in `OrderbookMarket` does not require `ACTIVE` status

File(s): `Orderbook/Market/OrderbookMarket.sol`

Description: The `_cancel()` function only checks that the market has not been resolved, but does not verify the market is in `ACTIVE` status:

```
function _cancel() internal {  
    if (resolution.resolved) revert MarketResolved();  
  
    OrderbookMarketStatus oldStatus = status;  
    status = OrderbookMarketStatus.CANCELLED;  
  
    emit StatusChanged(oldStatus, status);  
    emit MarketCancelled();  
}
```

This means a market that is already `PAUSED` or `CANCELLED` can be cancelled again. In the case of re-cancellation, the function succeeds silently (changing `CANCELLED` to `CANCELLED`), consuming gas and emitting duplicate `MarketCancelled` events. While this does not affect contract state, it is inconsistent with the expectation that cancel is a one-way state transition from `ACTIVE` (or `PAUSED`) to `CANCELLED`.

Recommendation(s): Consider adding a status check to ensure the market is in a cancellable state (e.g., `ACTIVE` or `PAUSED`) before allowing cancellation.

Status: Fixed

Update from the client: Fixed in [PR](#)

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Palpitada's documentation

Palpitada's team provided an overview of the main system components during audit calls with a detailed explanation of the intended functionalities and use cases of the system. Moreover, the team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Tests Output

```

npm test

> test
> npx hardhat test

Compiled 127 Solidity files successfully (evm target: istanbul).

CreditManager
  Deployment
    Should deploy with correct owner
  Authorization
    Should allow market factory to set an authorized consumer
    Should revert if trying to authorize zero address
    Should revert if non-market-factory tries to authorize
  Market Factory
    Should allow setting market factory through multisig
    Should revert if trying to set zero address as market factory
    Should revert if non-multisig tries to execute setMarketFactory
  Credits
    Should revert when non-multisig tries to add credits
    Should allow adding credits through multisig
    Should track credit balance correctly
    Should allow consuming credits through multisig
    Should revert if user has insufficient credit
    Should revert when trying to consume credits from zero address
    Should revert if non-multisig tries to consume credits
    Should revert when adding zero or to zero address
    Should allow fractional USDC amounts (6 decimal precision)
    Should revert consumeCredit with zero amount
    Should allow consuming fractional USDC amounts
  Additional Coverage Tests
    Should revert setMultisigController if not owner (line 49)
    Should revert setMultisigController if already initialized (line 50)
    Should revert setMultisigController with zero address (line 51)
    Should allow owner to set market factory initially
    Should revert setMarketFactory if not owner
    Should revert setMarketFactory if already initialized
    Should handle consumeCredit selector in executeMultisigOperation (line 75)
    Should revert addCredit if called by unauthorized address
    Should revert consumeCredit if called by unauthorized address
    Should revert executeMultisigOperation with invalid selector

GovernanceToken tests
  Constructor
    Should correctly initialize the contract
  Permit (EIP-2612) Tests
    permit()
      Should allow an account to sign a permit and give allowance
      Should increase the nonce after using permit
      Should revert if the signature is invalid
      Should revert if permit is used after the deadline

Dynamic Liquidity Parameter (b) Tests
  Market Creation with Different b Values
    Should create market with default b (0 passed, uses DEFAULT_LIQUIDITY_PARAMETER)
    Should create market with custom b value
    Should create market with auto-dynamic b enabled
    Should revert with InvalidLiquidityParameter for b below minimum
    Should revert with InvalidLiquidityParameter for b above maximum
  Auto-Dynamic b Calculation
    Should calculate b based on pool size tiers
    Should NOT auto-calculate b when autoDynamicLiquidity is false
  Position-Aware Dynamic b
    Should use projected pool size for large positions on empty market
    Should calculate appropriate b for institutional position on empty market
    Should add position to existing pool for tier calculation
  Multisig Control of Dynamic b
    Should allow setting liquidity parameter via multisig

```

- Should allow toggling auto-dynamic liquidity via multisig
- Should revert setLiquidityParameter with invalid value via multisig
- Effect of b on Price Calculations
 - Should have lower slippage with higher b
 - Should **return** correct previewBuy data
- MarketCalculations Library Dynamic b
 - Should calculate dynamic b based on pool tiers (10x mapping)
 - Should correctly use b parameter **in** getCostFunction
 - Should correctly use b parameter **in** getPrice
 - Should correctly use b parameter **in** getOdds
 - Should calculate correct b at tier boundaries (micro and tiny) with 10x mapping
 - Should calculate correct b at tier boundaries (standard tiers) with 10x mapping
 - Should calculate correct b at tier boundaries (institutional tiers) with 10x mapping
 - Should handle edge **case** of zero pool size with 10x mapping
 - Should handle very small pool sizes (below \$100) with 10x mapping
 - Should maintain tier consistency within tier ranges with 10x mapping
- Pool Tier Constants Verification
 - Should have correct POOL_TIER constants defined
 - Should have correct MIN/MAX liquidity parameter constants
- Token Fee Discount System
 - Token Tier Configuration
 - Should configure PPT token with multiple tiers
 - Should configure partner token with lower tier discounts
 - Should remove token discount configuration
 - Fee Discount Application
 - Should apply no discount **for** users without tokens
 - Should apply correct discount based on token balance tier
 - Should use highest discount when user holds multiple tokens
 - Should use partner token discount **if** user doesn't have enough PPT
 - End-to-End Prediction with Discounts
 - Should correctly apply discounts **in** actual predictions
 - Sell Fee Discounts
 - Should apply discount to sell fees
 - Dynamic Tier Updates
 - Should update discount when user acquires more tokens
 - Should update discount when user loses tokens
 - Edge Cases
 - Should handle users with exactly threshold amounts
 - Should handle empty tier configurations
 - Should handle tokens with single tier
 - Token Discount Configuration Edge Cases
 - Should revert when configuring zero address token
 - Should revert when configuring empty tiers array
 - Should revert when discount exceeds 10000 basis points
 - Should revert when tier thresholds are not **in** ascending order
 - Should revert when tier thresholds are equal
 - Should revert when removing non-configured token
 - Should handle removing token from middle of discount tokens array
 - Should handle inactive token configurations correctly
- Liquidity-Based Fee Scaling
 - Fee threshold behavior
 - Should charge zero fees when liquidity is below \$100 on both sides
 - Should charge zero fees when only one side has liquidity
 - Should charge zero fees when both sides have liquidity but min side < \$100
 - Should charge full fees when both sides have >= \$100 liquidity
 - MIN_LIQUIDITY_FOR_FEES constant
 - Should have MIN_LIQUIDITY_FOR_FEES **set** to \$100 **in** 18 decimals
 - Fee collection after threshold is met
 - Should correctly accumulate protocol fees after threshold
 - Should not accumulate protocol fees before threshold
 - Edge cases
 - Should use minimum of both sides **for** threshold check
 - Should start charging fees immediately when threshold is crossed
- Market tests
 - Constructor
 - Should correctly initialize the contract
 - Should revert **if** multisig controller address is zero
 - Should revert **if** credit manager address is zero
 - Should revert **if** USDC token address is zero
 - Should revert **if** protocol fee equal or greater than 10000
 - Should revert **if** initial odds are zero
 - Should revert **if** initial odds don't sum to 100%

Should revert **if** the event date is invalid
 Should revert **if** the category is invalid
 Should revert **if** the subcategory is empty
 Should revert **if** the name is empty
 Should revert **if** the subtitle is empty
 Should revert **if** the description is empty
 Should revert **if** the rules are empty
 Should revert **if** the image is empty
 Should revert **if** trying to initialize twice
 Should revert **if** minLiquidityRatio is less than 1000
 Should revert **if** minLiquidityRatio is greater than 5000
 Should revert **if** baseSellFee exceeds 2000
 Should revert **if** minHoldingPeriod exceeds 7 days

Place Prediction

Should allow a user to place a prediction on Side A
 Should allow a user to place a prediction on Side B
 Should allow a user to place a prediction with credits and money
 Should allow a user to place a prediction with credits only
 Should revert **if** the event is settled
 Should revert **if** the event is closed
 Should revert **if** shares amount is zero
 Should revert **if** shares would exceed MAX_SHARES limit (exp overflow protection)
 Should revert **if** shares would exceed MAX_SHARES on side B
 Should revert **if** the permit deadline is expired

Market Place Prediction Tests with Different Initial Odds

Initial Odds A: 50% | Initial Odds B: 50%
 Should **set** initial prices correctly **for** Side A (5000000000000000) and Side B (5000000000000000)
 Initial Odds A: 60% | Initial Odds B: 40%
 Should **set** initial prices correctly **for** Side A (6000000000000000) and Side B (4000000000000000)
 Initial Odds A: 40% | Initial Odds B: 60%
 Should **set** initial prices correctly **for** Side A (4000000000000000) and Side B (6000000000000000)
 Initial Odds A: 70% | Initial Odds B: 30%
 Should **set** initial prices correctly **for** Side A (7000000000000000) and Side B (3000000000000000)
 Initial Odds A: 30% | Initial Odds B: 70%
 Should **set** initial prices correctly **for** Side A (3000000000000000) and Side B (7000000000000000)
 Initial Odds A: 80% | Initial Odds B: 20%
 Should **set** initial prices correctly **for** Side A (8000000000000000) and Side B (2000000000000000)
 Initial Odds A: 20% | Initial Odds B: 80%
 Should **set** initial prices correctly **for** Side A (2000000000000000) and Side B (8000000000000000)
 Initial Odds A: 90% | Initial Odds B: 10%
 Should **set** initial prices correctly **for** Side A (9000000000000000) and Side B (1000000000000000)
 Initial Odds A: 10% | Initial Odds B: 90%
 Should **set** initial prices correctly **for** Side A (1000000000000000) and Side B (9000000000000000)
 Initial Odds A: 99% | Initial Odds B: 1%
 Should **set** initial prices correctly **for** Side A (9900000000000000) and Side B (1000000000000000)
 Initial Odds A: 1% | Initial Odds B: 99%
 Should **set** initial prices correctly **for** Side A (1000000000000000) and Side B (9900000000000000)

Set Winner

Should allow setting the winner as Side A via multisig
 Should allow setting the winner as Side B via multisig
 Should revert **if** the event is already settled
 Should revert **if** the event is not closed yet
 Should allow the owner to **set** the result as a draw

Withdraw

Should allow a user to withdraw their winnings **for** Side A
 Should allow a user to withdraw their winnings **for** Side B
 Should revert **if** prediction was lost **for** Side A
 Should revert **if** prediction was lost **for** Side B
 Should revert withdraw **if** the event is not settled
 Should **return** prediction amounts to users **in case** of a draw
 Should **return** credits amounts to users **in case** of a draw
 Should handle draw withdrawal when user paid 100% with credits (payout = 0)
 Should allow winner to withdraw when prediction was made with credits and USDC
 Should revert with NoPayoutAvailable when calculated payout is zero

Emergency Withdraw

Should allow the owner to withdraw all funds **in** an emergency
 Should reset curatorFeesBalance on emergency withdrawal

Deposit USDC

Should allow owner to deposit USDC
 Should revert when non-owner tries to deposit USDC
 Should revert **if** owner hasn't approved the market

Set Protocol Fee

Should allow the owner to **set** the protocol fee
 Should revert **if** a non-owner tries to **set** the protocol fee
 Should revert **if** the combined fee exceeds 100%

- Should revert **if** protocol fee is **set** to zero
- Get Odds
 - Should **return** the correct odds **for** a prediction on Side A
 - Should **return** the correct odds **for** a prediction on Side B
 - Should **return** 0 when price is 0 to avoid division by zero
- Preview Payout
 - Should **return** the correct payout **for** Side A
 - Should **return** the correct payout **for** Side B
 - Should **return** 0 payout **for** Side A when user has no shares
 - Should **return** 0 payout **for** Side B when user has no shares
- Withdraw Protocol Fees
 - Should allow the owner to withdraw protocol fees after market is settled
 - Should revert **if** a non-owner tries to withdraw protocol fees
 - Should revert **if** there are no protocol fees to withdraw
 - Should revert withdrawProtocolFees **if** the event is not settled
 - Should revert **if** the event is a draw
- Market LMSR Test Suite
 - Should correctly track predictions and adjust odds dynamically
 - Should distribute payouts correctly when Side A wins
 - Should distribute payouts correctly when Side B wins
 - Should revert predictions **if** event is already settled
- Market Closure Functionality
 - Should allow closing market through multisig
 - Should revert when trying to call closeMarket directly
 - Should revert predictions after market is closed
 - Should allow setting winner after market is closed early
 - Should not allow closing an already closed market
 - Should not allow closing a settled market
 - Should handle state transitions correctly
 - Should allow withdrawals after early closure and settlement
- Sell Position
 - Market Status Checks
 - Should revert **if** trying to sell when market is not open
 - Should emit LiquidityWarning when liquidity drops below threshold
 - Should **return** 100% liquidity ratio when no investments have been made
 - Basic Sell Functionality
 - Should allow users to sell their positions
 - Should enforce minimum holding period
 - Should reset holding period when fully exiting position
 - Should apply dynamic fees based on holding duration
 - Should prevent selling more shares than owned
 - Should revert **if** trying to sell zero shares
 - Solvency Protection
 - Should prevent sells that would breach minimum liquidity ratio
 - Should handle multiple sequential sells correctly
 - Should maintain solvency **for** winner payouts after sells
 - Admin Controls
 - Should allow admin to pause sells **in** emergency
 - Should allow admin to adjust sell parameters
 - Should reject invalid admin parameter updates
 - Edge Cases
 - Should handle sells when market has low liquidity
 - Should enforce dynamic max sell based on liquidity
 - Should prevent users who used credits from selling
 - Should correctly update position tracking after sells
 - Should **return** zeros when getSellReturn called with zero shares
 - Should **return** zeros when getSellReturn called with insufficient shares
 - Comprehensive Solvency Tests
 - Should maintain pool solvency when all winning positions sell
 - Should handle large sells at peak price correctly
 - Price Consistency Tests
 - Should **return** to original price after buy and sell of same amount
 - Should maintain price consistency across multiple buy/sell cycles
 - Fee Distribution Tests
 - Should split sell fees between protocol and predictions pool
 - Should handle totalCostB becoming zero when sell amount exceeds it
 - Total Cost Edge Cases
 - Should **set** totalCostA to 0 when sell amount exceeds it
 - Should properly handle totalCostB edge **case**
 - Edge Case Coverage Tests
 - Should handle fee cap at exactly 30%
 - Should **return** 0 odds when price is 0
 - Dynamic Max Sell Percentage Tests
 - Should allow 100% sell when side has 50%+ of pool
 - Should enforce 50% sell limit when side has 40-50% of pool

- Should enforce 25% sell limit when side has 30-40% of pool
- Should enforce 10% sell limit when side has 20-30% of pool
- Should enforce 5% sell limit when side has less than 20% of pool
- Should handle liquidity tier transitions smoothly
- Should calculate correct max sell **for** partial positions
- Should handle concurrent sells with dynamic limits
- Should respect dynamic limits **for** both sides
- Should prevent exploitation of dynamic limits
- Side-Specific Liquidity Tests
 - Should restrict sells when side has less than 20% of total pool
 - Should allow balanced sells when sides are equal
 - Should progressively restrict sells as side liquidity decreases
- Time-to-Event Dynamic Fees Tests
 - Should block all sells 3 days before event
 - Should apply maximum fees 4 days before event
 - Should apply increasing fees as event approaches
 - Should apply 3x fee **for** sides with less than 20% of pool
 - Should cap combined fees at 30%
 - Should enforce 30% fee cap even with extreme multipliers
- Combined Scenario Tests
 - Should handle complex scenario with multiple users and **time** progression
 - Should prevent death spiral when one side mass-exits
- Coverage Edge Cases
 - Should trigger **totalCostA** = 0 branch when sell exceeds cost
 - Should trigger **totalCostB** = 0 branch when sell exceeds cost
 - Should apply 1.5x fee multiplier exactly 6 days before event
- Attack Vector Tests
 - Should prevent sandwich attacks
 - Should prevent pool drainage attempts
 - Should handle concurrent buys and sells correctly
 - Should handle sells at minimum liquidity correctly
- Liquidity Edge Cases Coverage
 - Should revert with **InsufficientLiquidityForSells** when dynamic max sell percentage is 0
 - Should revert with **LiquidityBreach** when sell would violate minimum liquidity ratio
 - Should revert with **InsufficientShares** when user has no winning position
 - Should revert with **InsufficientContractFunds** **if** contract doesn't have enough USDC
- Credit Usage Tracking
 - Should track credits used per side correctly
 - Should emit **CreditUsageTracking** event when credits are used
 - Should track zero credits when prediction is made with USDC only
 - Should reset credit tracking on emergency withdrawal
- Access Control and Edge Cases
 - executeMultisigOperation** from non-controller
 - Should revert when **executeMultisigOperation** is called by non-controller
 - sellPosition** after event date
 - Should revert when trying to sell position after event date
 - Sell configuration functions - Owner only
 - Should revert when trying to call **setSellsEnabled** directly
 - Should allow calling **setSellsEnabled** through multisig
 - Should revert when trying to call **setMinLiquidityRatio** directly
 - Should allow calling **setMinLiquidityRatio** through multisig
 - Should revert when trying to call **setBaseSellFee** directly
 - Should allow calling **setBaseSellFee** through multisig
 - Should revert when trying to call **setMinHoldingPeriod** directly
 - Should allow calling **setMinHoldingPeriod** through multisig
- Curator Fee System - Coverage Tests
 - Should reject curator fee over 5% (500 basis points)
 - withdrawCuratorFees** **function** coverage
 - Should allow curator to withdraw fees after settlement
 - Should revert **if** non-curator tries to withdraw curator fees
 - Should revert curator withdrawal before settlement
 - Should revert **if** curator has no fees to withdraw
 - Should revert curator fee withdrawal on draw (users need full refunds)
 - Curator fee collection during predictions
 - Should collect curator fees when curator is **set** and fee > 0
 - Should not collect curator fees when curator is zero address
 - Should not collect curator fees when curatorFee is 0
- MarketCalculations - Inactive Config Coverage
 - Should skip inactive token configs when calculating discounts
 - Should correctly handle mix of active and inactive configs
- MarketFactory tests
 - Constructor
 - Should correctly initialize the contract
 - Should revert **if** the USDC token is zero address

```

    Should revert if the market implementation is zero address
    Should revert if the credit manager is zero address
    Should revert if the multisig factory is zero address
Create Market
    Should create a new Market contract
    Should revert if multisig signers array is empty
    Should revert if not called by account with MARKET_CREATOR_ROLE
    Should allow account with MARKET_CREATOR_ROLE to create market
    Should allow admin to revoke MARKET_CREATOR_ROLE
    Should keep track of all created Market contracts
MultisigController One-Time Integration
    Should create a MultisigController for each market
    Should configure MultisigController with correct signers and threshold (one-time configuration)
    Should keep MultisigControllerFactory as owner of MultisigController
    Should prevent reconfiguration of MultisigController (enforces one-time configuration per market)
    Should prevent re-authorization of targets (one-time configuration only)
    Should prevent deauthorization of configured targets (configuration is permanent)

Market Reentrancy Protection Tests
Predict Function Reentrancy
    Should prevent reentrancy attack on predict function via malicious token
    Should emit ReentrancyAttempted event with success=false when attack is blocked
Withdraw Function Reentrancy
    Should prevent reentrancy attack on withdraw function
SellPosition Function Reentrancy
    Should prevent reentrancy attack on sellPosition function
Multiple Reentrancy Attempts
    Should handle multiple reentrancy attempts gracefully

Slippage Analysis Tests (Appendix A Examples)
A.1: Fresh Market, 50/50 odds, b=10,000
$100 position: slippage = 0.49%
    $100 position at b=10,000: ~0.5% slippage
$500 position: slippage = 2.49%
    $500 position at b=10,000: ~2.5% slippage
$1,000 position: slippage = 4.99%
    $1,000 position at b=10,000: ~5% slippage
$2,500 position: slippage = 12.37%
    $2,500 position at b=10,000: ~12.37% slippage
$5,000 position: slippage = 24.02%
    $5,000 position at b=10,000: ~24.02% slippage
$10,000 position: slippage = 43.37%
    $10,000 position at b=10,000: ~43.37% slippage
A.1 continued: Same positions with b=100,000 (10x)
b=100k, $100 position: slippage = 0.040%
    $100 position at b=100,000: ~0.04% slippage (10x less than b=10k)
b=100k, $1,000 position: slippage = 0.49%
    $1,000 position at b=100,000: ~0.49% slippage (10x less than b=10k)
b=100k, $10,000 position: slippage = 4.99%
    $10,000 position at b=100,000: ~5% slippage (10x less than b=10k)
A.3: b=1,000,000 (Institutional Grade)
b=1M, $10,000 position: slippage = 0.490%
    $10,000 position at b=1,000,000: ~0.49% slippage (100x less than b=10k)
b=1M, $50,000 position: slippage = 2.49%
    $50,000 position at b=1,000,000: ~2.5% slippage
b=1M, $100,000 position: slippage = 4.99%
    $100,000 position at b=1,000,000: ~5% slippage
A.4: Imbalanced Market (70/30 Odds)
Favorite (A) quoted price: $0.700
70/30, $1,000 on favorite: slippage = 2.10%
    70/30 market: $1,000 on favorite (70%): ~2.1% slippage
70/30, $5,000 on favorite: slippage = 9.61%
    70/30 market: $5,000 on favorite (70%): ~9.61% slippage
Underdog (B) quoted price: $0.300
70/30, $1,000 on underdog: slippage = 12.15%
    70/30 market: $1,000 on underdog (30%): ~12.15% slippage (higher than favorite)
70/30, $5,000 on underdog: slippage = 65.56%
    70/30 market: $5,000 on underdog (30%): ~65.56% slippage (much higher than favorite)
$1000 on favorite: 2.10% slippage
$1000 on underdog: 12.15% slippage
KEY INSIGHT: Underdog has HIGHER slippage than favorite for same dollar amount
A.5: Extreme Odds (90/10)
Favorite (A) quoted price: $0.900
90/10, $1,000 on favorite: slippage = 0.53%
    90/10 market: $1,000 on favorite (90%): ~0.53% slippage

```

Underdog (B) quoted price: \$0.100
 90/10, \$1,000 on underdog: **slippage** = 58.55%
 90/10 market: \$1,000 on underdog (10%): ~58.55% slippage (EXTREME)
 90/10, \$5,000 on underdog: **slippage** = 451.23%
 90/10 market: \$5,000 on underdog (10%): ~451.23% slippage (4.5x quoted price!)
 90/10, \$10,000 on underdog: **slippage** = 669.76%
 90/10 market: \$10,000 on underdog (10%): ~669.76% slippage (6.7x quoted price!)

A.6: Slippage Protection Integration
 Should revert when average price exceeds maxPricePerShare
 Should succeed with appropriate slippage tolerance
 Should bypass slippage check when maxPricePerShare = 0

A.7: Key Insight - 10x b = ~10x slippage reduction
 \$1K at b=10k: 4.99%
 \$1K at b=100k: 0.490%
 \$1K at b=1M: 0.0400%
 Improvement ratio b=10k+100k: 10.2x
 Improvement ratio b=100k+1M: 12.3x
 VALIDATES: 10x b **improvement** = ~10x slippage reduction
 \$10K at b=10k: 43.37%
 \$10K at b=100k: 4.99%
 \$10K at b=1M: 0.490%
 \$10K position: slippage scales with b parameter

Institutional Arbitrage Viability
 \$1K at b=10k: 4.99% slippage
 \$100K at b=1M: 4.99% slippage
 b=1M enables larger positions than b=10k at similar slippage
 b=10k, \$10K entry slippage: 43.37%
 b=10k makes \$10K+ positions expensive (high slippage)

Slippage Protection Tests
 previewBuy Function
 Should **return** accurate cost, averagePrice, quotedPrice, and slippageBps
 Should show increasing slippage **for** larger trades
 Should preview both sides correctly

Slippage Protection **in** predict()
 Should succeed when average price is below maxPricePerShare
 Should revert with SlippageExceeded when average price exceeds maxPricePerShare
 Should bypass slippage check when maxPricePerShare is 0
 Should protect against front-running attacks
 Should work with exact average price as maxPricePerShare

Edge Cases
 Should handle very small trades (1 share)
 Should handle trades near the initial state
 Should handle trades after market has moved
 Should correctly calculate slippage **for** both sides

Integration with Dynamic b
 Should have lower slippage with higher liquidity parameter

Slippage Events and Error Data
 Should include correct values **in** SlippageExceeded error

Stress Tests
 Should handle rapid consecutive trades with slippage protection
 Should reject trades when market moves significantly between preview and execution

MultisigController Tests
 MultisigController One-Time Setup
 Should correctly configure multisig **for** a target (one-time only)
 Should have MultisigControllerFactory as owner
 Should prevent reconfiguration of multisig (enforces one-time configuration)
 Should revert with TargetAlreadyConfigured when trying to reconfigure

setWinner via Multisig
 Should execute setWinner with valid multisig signatures
 Should fail with insufficient signatures
 Should only allow setWinner via multisig (no direct access)

emergencyWithdraw via Multisig
 Should execute emergencyWithdraw with valid multisig signatures

setProtocolFee via Multisig
 Should execute setProtocolFee with valid multisig signatures
 Should prevent replay attacks
 Should fail with expired deadline

Per-Function Thresholds
 Should use **function**-specific threshold when configured
 Should fail emergencyWithdraw with only 2 signatures when 3 are required
 Should succeed emergencyWithdraw with all 3 signatures

Delayed Operations
 Should queue operation when delay is configured

- Should fail execution before delay period
- Should execute successfully after delay period
- Cancel Operation via Multisig
 - Should handle cancel operation selector **for** self-targeting operations
- Credit Functions via Multisig
 - Should add credits through multisig
 - Should consume credits through multisig
- Removed External Functions
 - Should not have direct access to admin functions **in** Market
 - Should have setMarketFactory as owner-only **for** initial setup **in** CreditManager
 - Should execute market admin functions only through multisig
- Target Configuration Requirements
 - Should not execute operations on targets that haven't been configured (one-time config required)
 - Should verify target has been configured (one-time configuration completed)
- One-Time Configuration Validation
 - Should reject configuration with zero address target
- One-Time Configuration State Tracking
 - Should correctly track one-time configuration state per target
 - Should maintain separate one-time configuration state **for** different targets
- Configuration Edge Cases
 - Should handle configuration with minimum signers
 - Should handle configuration with maximum reasonable signers
- Invalid Selector Coverage
 - Should revert with InvalidSelector **for** unknown **function** selector
- Invalid Configuration Parameters
 - Should revert when threshold is 0
 - Should revert when threshold exceeds signers length
 - Should revert when signers array is empty
 - Should verify empty signers array handling (defensive programming)
 - Should revert when a signer is zero address
 - Should revert when signers array contains duplicates
 - Should revert when per-function threshold exceeds signers length
 - Should allow per-function threshold of 0 (uses default)
- Operation Replay Prevention
 - Should verify operation execution tracking (defensive programming)
- Signature Validation
 - Should revert when signature is from non-signer
 - Should revert when same signer signs twice
 - Should prevent unauthenticated queuing of delayed operations
- Target Execution
 - Should revert when target contract execution fails
- MultisigControllerFactory
 - Should revert when non-owner tries to create MultisigController
- Additional Coverage Tests
 - Should handle immediate execution path (line 106 **else** path)
 - Should **test** getOperationDelay **function**
 - Should **test** pending operations tracking (line 151 coverage)
 - Should properly queue and execute delayed operation
 - Should handle different threshold requirements **for** different functions
 - Should revert with InvalidSelector **for** non-cancelOperation on controller itself (line 107)
 - Should properly **test** getOperationDelay **function**
 - Should revert with OperationNotQueued when trying to cancel non-queued operation (line 151)
- Real-World Multisig Scenarios
 - Complete Emergency Withdraw Flow
 - Should execute emergency withdraw with proper multisig and delay
 - Complete Cancel Operation Flow
 - Should cancel a queued operation through proper multisig
 - Credit Management Through Multisig
 - Should add and consume credits with proper thresholds and delays
 - Market Admin Operations
 - Should close market and **set** winner through multisig
 - Security and Edge Cases
 - Should prevent execution with insufficient signatures **for** specific functions
 - Should prevent replay attacks with nonce management
 - Should verify removed external functions cannot be called directly
- OrderbookExchange - Redeem Functions
 - executeRedeem - Resolved Markets
 - should execute redeem **for** user with winning tokens
 - should revert **if** market is neither resolved nor cancelled
 - should revert **if** signature is invalid
 - should revert **if** nonce is invalid
 - should revert **if** order already executed
 - should increment nonce after successful redeem

```

    should revert if user has no winning tokens
executeRedeem - Cancelled Markets
    should execute redeem for cancelled market with tokens
View Functions
    should return correct redeem order hash
    should validate redeem order correctly
Access Control
    should revert executeRedeem if not operator

OrderbookExchange
  Initialization
    should initialize correctly
    should revert with zero USDC address
    should revert with zero operator address
    should revert with zero multisig controller address
    should revert with zero factory address
    should revert if already initialized
  Market Registration
    should allow factory to register market
    should not allow non-factory to register market
    should not allow registering same market twice
    should allow unregistering market via multisig
  Admin Functions
    should allow changing operator via multisig
    should not expose setOperator as external function (multisig only)
    should allow changing fee recipient via multisig
    should not allow setting zero address as operator via multisig
  Order Validation
    should validate a correctly signed order
    should reject order with invalid signature
    should reject order for unregistered market
  Order Hashing
    should compute consistent order hash
    should produce different hashes for different orders
  DIRECT Match (BUY vs SELL)
    should execute direct match transferring tokens
    should revert if orders have same position (same side, same tokenId)
    should revert DIRECT trade with mismatched tokenIds
  MINT Match (BUY + BUY)
    should mint new tokens from two BUY orders
  MERGE Match (SELL + SELL)
    should merge tokens from two SELL orders
    should merge tokens when taker sells NO position
  Fee Handling
    should accumulate fees from matches
    should allow withdrawing accumulated fees via multisig
    should not expose withdrawFees as external function (multisig only)
    should revert withdrawFees with zero balance via multisig
    should correctly transfer fees to Exchange from MINT operations
    should correctly transfer fees to Exchange from MERGE operations
    should have matching accumulatedFees and Exchange USDC balance after multiple operations
  Access Control
    should only allow operator to match orders
  Edge Cases
    should revert on zero fill amount
    should revert when markets mismatch
  DOMAIN_SEPARATOR
    should return the EIP-712 domain separator
  Partial Fills
    should allow same signed order to be used in multiple partial fills (MINT)
    should allow same signed order to be used in multiple partial fills (DIRECT)
    should allow partial fill with amount smaller than order's makerAmount
    should revert when attempting to overfill an order
    should track filled amounts correctly across partial fills
  Ternary Market Three-Way Matching
    should verify ternary market has hasDraw enabled
    should execute ternary MINT with three BUY orders
    should execute ternary MERGE with three SELL orders
    should revert with draw order on binary market
    should revert with invalid token IDs for ternary MINT
    should revert with mixed sides for ternary orders
    should revert with zero fill amount for ternary orders
    should allow DIRECT trade of DRAW tokens in ternary market
    should only allow operator to call fillOrders with ternary orders
  Multisig Operations - Additional Coverage

```

```

    should allow changing factory via multisig
    should allow pausing via multisig
    should allow unpausing via multisig
    should revert with invalid selector
isOrderValid - Edge Cases
    should return false for order on inactive market
    should return false when market.isActive() throws
    should return false for order with invalid signature
100% Coverage - Missing Branches
Paused State Tests
    should revert fillOrders when exchange is paused
    should revert executeMultisigOperation from non-multisig caller
Zero Address Validation Tests
    should revert initialize with zero feeRecipient
    should revert _registerMarket with zero address via factory
    should revert setFeeRecipient with zero address via multisig
    should revert setFactory with zero address via multisig
Unregister Market Edge Case
    should revert unregistering a non-existent market
DIRECT Match - SELL Side
    should execute DIRECT match when taker sells
MINT Match - NO Position (Taker Wants NO)
    should execute MINT when taker wants NO position
Edge Cases for 100% Coverage
Zero Fee Paths - Small Amounts
    should handle MINT with very small amounts resulting in zero fees
Order Validation Edge Cases
    should revert expired order in fillOrders
    should handle invalid taker restriction
Ternary Market Token ID Validation
    should revert with mismatched token IDs in ternary market
Signature Validation Edge Cases
    should return false for malformed signature in isOrderValid
    should return false for signature with invalid v value
fillOrders Signature Validation
    should revert fillOrders with malformed signature
Market Validation Error Paths
    should revert fillOrders with unregistered market
    should revert fillOrders with inactive market

OrderbookFactory
Constructor Validation
    should revert when usdcToken is zero address
    should revert when marketImpl is zero address
    should revert when multisigFactory is zero address
    should deploy successfully with valid parameters
createMarket Validation
    should revert when exchange not set
    should revert when marketData.name is empty
    should revert when endTime is in the past
    should revert when protocolFee exceeds 10000
    should revert when multisigSigners array is empty
    should revert when multisigThreshold is zero
    should revert when multisigThreshold exceeds signers length
    should create market successfully with valid parameters
Access Control
    should revert when non-admin calls setExchange
    should revert setExchange with zero address
    should allow admin to set exchange
    should revert when non-admin calls grantMarketMaker
    should revert grantMarketMaker with zero address
    should allow admin to grant market maker role
    should revert when non-admin calls revokeMarketMaker
    should allow admin to revoke market maker role
    should revert when non-creator calls createMarket
View Functions
    should return all markets via getMarkets
    should return correct market count via getMarketCount
    should return multisig controller for created market
Market Creation Integration
    should create binary market correctly
    should create ternary market correctly
    should create ternary market with custom sideC and imageC

```

Market Maker Flows

Suite 1: Orderbook Spread Capture
 should capture orderbook spread (BUY low, SELL high)
 should handle partial fills on spread capture

Suite 2: Split Arbitrage
 should execute split arbitrage when YES + NO > \$1.00
 should calculate fees correctly on split arbitrage

Suite 3: Merge Arbitrage
 should execute merge arbitrage when YES + NO < \$1.00
 should handle merge arbitrage with minimal spread

Suite 4: Edge Cases and Complex Scenarios
 should prevent profit when YES + NO = \$1.00
 should handle round-trip with no price movement
 should execute multiple arbitrage cycles

Suite 5: Ternary Market MM Flows
 should execute split arbitrage on ternary market
 should execute merge arbitrage on ternary market

MM ECONOMIC VALIDATION SUMMARY

spreadCapture

Test ROI: 5.63%
 Net Profit: \$27.05
 Fee Burden: 249.0%
 Not viable
 ROI 5.63% barely above minimum 5%
 Fee burden 249.0% exceeds maximum 40%

splitArbitrage

Test ROI: 12.20%
 Net Profit: \$122.01
 Fee Burden: 0.0%
 All thresholds met

mergeArbitrage

Test ROI: 15.21%
 Net Profit: \$129.28
 Fee Burden: 0.0%
 All thresholds met

Test results saved to:

→ /Users/dic0de/Desktop/2026/Feb/palpitada/v2/NM-0739-Security-Review-Palpitada/mm-test-results.json
 Run detailed analysis: npm run validate:mm

OrderbookMarket

Initialization

should initialize correctly
 should not allow double initialization

Resolution and Redemption

should allow multisig controller to **set** winner
 should not allow non-multisig-controller to **set** winner
 should allow redemption of winning tokens
 should allow NO holder to redeem when NO wins via redeemFor
 should not allow redemption via redeemFor before resolution
 should not allow DRAW resolution on binary market

Exchange Functions

should allow exchange to transfer tokens
 should not allow non-exchange to transfer
 should allow exchange to mint tokens
 should not allow non-exchange to mint tokens
 should allow exchange to burn tokens
 should revert burn with insufficient YES balance
 should revert burn with insufficient NO balance
 should allow exchange to transfer NO tokens
 should revert transfer YES with insufficient balance
 should revert transfer NO with insufficient balance
 should revert transfer DRAW on binary market
 should allow exchange to transfer DRAW tokens **in** ternary market
 should revert transfer DRAW with insufficient balance **in** ternary market
 should revert burn **in** ternary market with insufficient DRAW balance

Admin Functions

```

    should allow multisig controller to pause market
    should allow multisig controller to unpause market
    should allow multisig controller to cancel market
    should not allow non-multisig-controller to admin functions
    should prevent operations when market is paused
    should allow multisig controller to execute emergency withdrawal
    should revert emergency withdrawal with zero address recipient
    should revert emergency withdrawal with zero balance
    should not allow non-multisig-controller to emergency withdraw
    should revert with invalid selector
Cancellation Redemption
    should allow redemption after cancellation via redeemFor
    should return USDC with fixed equal distribution on cancellation
View Functions
    should return correct total supply after minting
    should return correct total reserve
    should return correct protocol fee
Market Maker Functions
    should allow market maker to split USDC into tokens
    should allow market maker to merge tokens back into USDC
    should not allow non-market-maker to split
    should not allow non-market-maker to merge
    should revert split with zero amount
    should revert merge with zero amount
    should revert merge with insufficient YES balance
    should allow revoking market maker role
    should not charge fees on split/merge (round trip)
    should not allow split when market is paused
    should not allow split when market is resolved
Ternary Market (DRAW-enabled)
    should initialize as ternary market with hasDraw=true
    should mint YES, NO, and DRAW tokens on split
    should require all three tokens for merge in ternary market
    should allow DRAW resolution in ternary market
    should allow DRAW holder to redeem when DRAW wins via redeemFor
    should revert merge with insufficient DRAW balance
    should handle cancellation redemption with all three tokens via redeemFor
Branch Coverage - Edge Cases
    should revert initialize with zero USDC address
    should revert initialize with zero exchange address
    should revert initialize with zero factory address
    should revert initialize with zero multisig controller address
    should revert redeemFor when user has zero winning balance
    should revert redeemFor when market is not resolved or cancelled
    should revert setWinner when already resolved
    should revert setWinner DRAW on binary market
    should revert pause when market is not active
    should revert unpause when market is not paused
    should handle ternary market DRAW redemption via redeemFor
    should handle ternary market SIDE_B redemption via redeemFor
    should revert merge in ternary market with insufficient DRAW balance
Branch Coverage - Edge Cases for 95%
    should revert merge when user has insufficient NO tokens
    should revert burn when holder has insufficient NO tokens
    should revert redeemFor when user has no tokens on cancelled market
    should revert cancel when market is already resolved
    should handle redeemFor when totalUsdc is zero on cancelled market
    should handle burn with zero proceeds
    should handle invalid tokenId in transferFrom with event but no transfer
Branch Coverage - State Modifiers
    should revert split when market is resolved
    should revert merge when market is resolved
    should revert split when market is paused
    should revert merge when market is paused
    should revert burn when market is paused
    should revert mint when market is resolved
Market Expiration (whenNotExpired modifier)
    should revert split when market is expired
    should revert merge when market is expired
    should revert transferFrom when market is expired
    should revert mint when market is expired
    should revert burn when market is expired
    should allow operations right before expiration
    should still allow redeemFor after expiration when resolved

```

```

OrderbookMarket - Split/Merge Access Control
  Permissionless Market (restrictSplitMerge = false)
    should allow any user to split USDC into tokens
    should allow any user to merge tokens back to USDC
    should allow multiple users to split simultaneously
    should verify splitMergeRestricted is false
  Permissioned Market (restrictSplitMerge = true)
    should allow market maker to split USDC into tokens
    should NOT allow regular user to split USDC
    should allow market maker to merge tokens back to USDC
    should NOT allow regular user to merge tokens
    should verify splitMergeRestricted is true
    should verify market maker has the role
    should verify regular user does NOT have market maker role
  MarketCreated Event
    should emit MarketCreated with restrictSplitMerge = false for permissionless market
    should emit MarketCreated with restrictSplitMerge = true for permissioned market

Orderbook - Replay Protection
  Order Hash Uniqueness
    should generate unique hash for each order with different salt
    should generate same hash for identical orders
    should include nonce in order hash
  Order Validation
    should accept valid order with correct signature
    should reject order with invalid signature
    should accept orders with any nonce value
  Order Fill Tracking
    should track order fills to prevent replay
  Split/Merge Nonces
    should have separate nonce for split orders
    should have separate nonce for merge orders
  Integration with Other Features
    should work correctly with taker restrictions
    should work correctly with order expiration
  Replay Attack Protection - Regular Orders (MINT)
    should prevent replay of fully filled MINT orders
    should prevent replay of partially filled orders beyond their limit
  Replay Attack Protection - DIRECT Orders
    should prevent replay of fully filled DIRECT orders
  Replay Attack Protection - Split Orders
    should prevent replay of executed split orders
    should prevent replay with wrong nonce
  Replay Attack Protection - Merge Orders
    should prevent replay of executed merge orders
    should prevent replay with wrong nonce
  Cross-Order Replay Attack Protection
    should not allow using split signature for merge
    should track fills independently for each order hash

OutcomeToken - Access Control
  Mint Access Control
    should allow market to mint
    should revert mint from non-market address
  Burn Access Control
    should allow market to burn
    should revert burn from non-market address
  Token Metadata
    should return correct token name and symbol
    should have 6 decimals (matching USDC)

645 passing (11s)

```

8.2 Automated Tools

8.2.1 AuditAgent

The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at <https://app.auditagent.nethermind.io>.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our cryptography Research team conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to [Nethermind](#) in order that [Nethermind](#) could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. [Nethermind](#) has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, [Nethermind](#) disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. [Nethermind](#) does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and [Nethermind](#) will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.