hexens x FERMAH

Security Review Report for Fermah

October 2026

Table of Contents

1. About Hexens
2. Executive summary
3. Security Review Details
    - Security Review Lead
    - Scope
    - Changelog
4. Severity Structure
    - Severity characteristics
    - Issue symbolic codes
5. Findings Summary
6. Weaknesses
    - Resolving a market during the betting window allows front-running of the outcome
    - Seed provider can make risk-free underdog bets at the expense of the treasury
    - Yield is shared by stake size only, so a late stake can take most of the accrued yield risk-free
    - Base's Transaction Gas Cap Prevents Ordinary Buys in 32-Outcome Markets
    - Protocol resolver fee ignores collateral decimals
    - Parlay baskets add too much liquidity depth: each selected cell is counted, though only one can pay
    - priceBps() is documented as the marginal price but returns the softmax probability
    - Seeded parlays refund losing bets when the winning cell is empty
    - Seed-cover scaling can reduce an earlier winner’s payout after a later bet
    - Seed is deposited into the yield vault, contrary to the documentation
    - Sub-basis-point prices cap fills while charging the full stake
    - Outstanding allowlist signature can override the owner's later direct update
    - Unused _isScalar() function
    - Dispute incentives weaken as the market grows
    - Misplaced NatSpec for _credit()

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This audit reviews Fermah Strata, a liquidity-sensitive LMSR market. A Strata market implements a liquidity-sensitive logarithmic market scoring rule (LS-LMSR) over N mutually exclusive outcomes. It operates on a buy-only basis: participants spend stablecoins to acquire shares in a particular outcome, with each outcome's price representing its marginal probability. Upon resolution, each winning share can be redeemed for one unit of collateral, less a volume-scaled settlement fee.

The audit was conducted over two weeks and involved a comprehensive review of the relevant smart contracts.

During the assessment, we identified one high-severity issue that allows an attacker to observe a market resolution transaction and front-run it by purchasing shares in the winning outcome to capture profits. In addition, we identified one medium-severity issue, three low-severity issues, and ten informational issues.

All identified issues were either remediated or acknowledged by the development team, and the corresponding fixes were subsequently verified by our auditors.

Following the remediation phase, we conclude that the audit has contributed to improving the protocol's overall security posture and code quality.

3. Security Review Details

Review Led by

Trung Dinh, Lead Security Researcher

Scope

The analyzed resources are located on:

https://github.com/fermah-labs/strata/tree/989ef62e3db320d56d34f9538c455b5f480eee5d

The issues described in this report were fixed in the following commit:

https://git.fermah.xyz/fermah-labs/strata/tree/449fdaf596feaf4392101fe1ca5b0e6a38520822

Changelog

28 September 2026	Audit start
13 October 2026	Initial report
19 October 2026	Revision received
13 November 2026	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

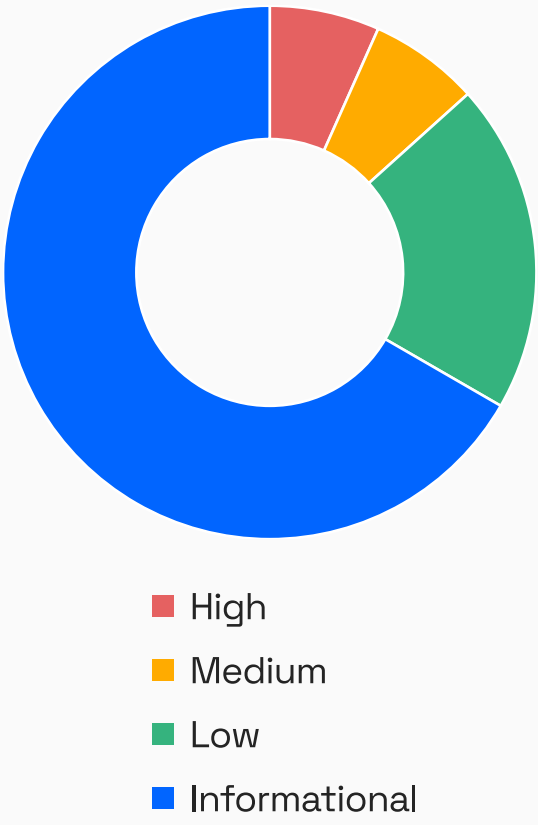
- **Issue Symbolic Codes**

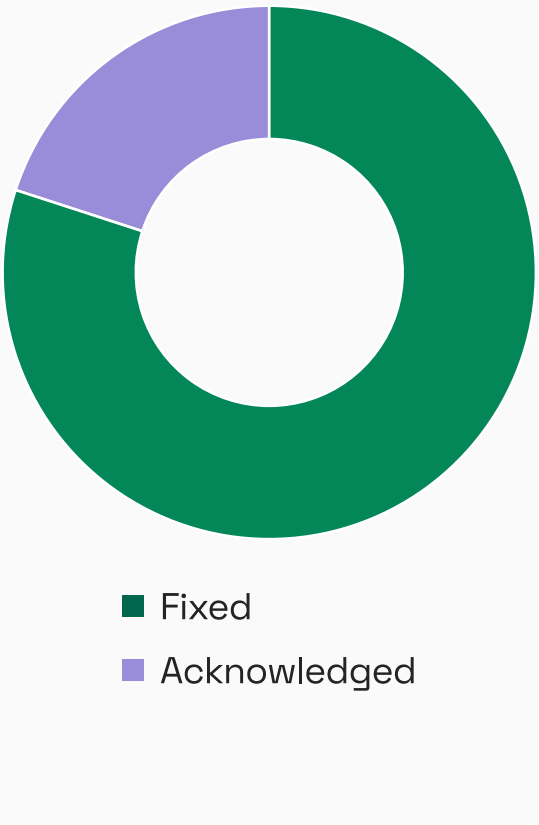
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	0
High	1
Medium	1
Low	3
Informational	10
Total:	**15**





6. Weaknesses

This section contains the list of discovered weaknesses.

FERM1-1 | Resolving a market during the betting window allows front-running of the outcome

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

contracts/src/MarketCore.sol#L1028-L1041

Description:

A market is resolved when its **resolver** calls one of **resolve()**, **resolveScalar()** or **resolveJoint()**. All of them end in **_settle()**, which marks the market resolved and fixes the winning inventory:

```
function resolve(
    uint256 winner
) external nonReentrant {
    if (msg.sender != resolver) {
        revert OnlyResolver();
    }
    if (_hasScalarLeg()) {
        revert InvalidResolutionKind();
    }
    _resolveCategorical(winner);
}
```

None of these functions check **closeTime**. Meanwhile, **_credit()** only stops a buy once the market is resolved or **closeTime** has passed:

```
if (resolved) {
    revert AlreadyResolved();
}

if (block.timestamp >= closeTime) {
    revert MarketClosed();
}
```

As a result, when a market is resolved before its **closeTime**, the resolving transaction is the first public disclosure of the outcome. Anyone who sees it can place a buy on the winning outcome ahead of it. The gasless **reportWithAuthorization()** path makes this easier: the relayer holds the signed outcome and can buy first, then submit the signature.

The front-runner buys the winning shares at the pre-resolution LMSR price and redeems them right after at close to 1 collateral per share.

Note that **DisputeResolver._openProposal()** already accounts for this and pulls **closeTime** in before publishing a proposal. However, that happens in the same transaction, so a buy ordered before the **propose** transaction is still possible, and the attackers can still benefit from it.

Remediation:

Consider requiring that betting has already closed before a non-void resolution is accepted, so that the resolver must stop betting in an earlier transaction (via **setCloseTime()**) before revealing the outcome:

```
function _settle(
    uint256[] memory weights
) internal {
++  if (block.timestamp <= closeTime && _weightedShares(_q) != 0) {
++      revert MarketStillOpen();
++  }
    resolved = true;
    ...
}
```

resolveVoid() can stay exempt, since a buy placed right before a void is refunded at par and gives no advantage. Closing betting and reporting in the same transaction does not prevent the issue, because the front-running buy executes earlier in the block while **closeTime** is still in the future.

FERM1-6 | Seed provider can make risk-free underdog bets at the expense of the treasury

Acknowledged

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

contracts/src/MarketCore.sol#L1390-L1396

Description:

The seed provider can make an underdog bet with no additional downside and use it to extract revenue that would otherwise go to the treasury and creator.

The seed provider is the sponsored funder that posted the seed, or the creator of a self-funded market (in which case it only gives up its own **creatorBps** share of the house take).

The issue is that the provider's losing bet is used to refill its own seed before the remaining funds are split between the treasury and creator. When the favourite's winners are owed more than the pot, the 95% floor in **_setSeedScale()** caps their haircut and the seed pays the remaining shortfall. **_sweep()** then refunds the seed from the recoverable balance, which includes losing stakes, before the house take is split:

```
uint256 seedRefund = Math.min(_pendingSeedRefund(), recoverable);
...
swept = recoverable - seedRefund;
uint256 creatorCut = Math.mulDiv(swept, creatorBps, BPS_DENOM);
uint256 treasuryCut = swept - creatorCut;
```

This could create the following outcome:

- If the favourite wins, the provider's losing bet fills the seed shortfall that the seed would have paid anyway, so the bet costs it nothing (up to the size of that shortfall).
- If the underdog wins, the provider receives its full seed back plus the betting payout, while the treasury and creator receive less house revenue.

The provider therefore gets a free option on the underdog, funded by the house.

For example, consider a binary market with alphaBps = 300, bMin = 10,000 USDC (seed 10,000 USDC) and the default halfTaper, so b = bMin throughout:

1. Bob bets 13,000 USDC on YES and receives 19,725.37 shares. Carol then bets 4,000 USDC on NO and receives 13,636.99 shares. The pot is 17,000 USDC and YES is priced at 60.40%.

2. If YES wins, the winners are owed **19,725.37 * 0.9803 = 19,336.78** (**keepBps = 9,803** from the fee curve at a 17,000 USDC volume with the default **halfTaper**), which exceeds the pot. **seedScaleBps** is floored at 9,500, so the winners receive 18,369.94 and the seed covers the 1,369.94 shortfall.
3. Just before the market closes, the seed provider bets 1,000 USDC on NO from another address and receives 2,402.90 shares.

The resulting outcomes are:

	Without the provider's bet	**With the 1,000 USDC NO bet**
YES wins: seed refund	8,630.06	9,630.06
YES wins: provider P&L	−1,369.94	−1,369.94 (the 1,000 loss for NO bet is offset by a 1,000 higher seed refund)
NO wins: provider payout	0	2,355.56
NO wins: provider P&L	0	+1,355.56
NO wins: house take	3,631.66	2,276.09

If YES wins, the provider's 1,000 USDC NO bet loses and enters the pot. This reduces the seed's shortfall one-for-one, from 1,369.94 to 369.94, so the provider loses its 1,000 USDC bet but gets exactly 1,000 USDC more of its seed back.

If NO wins, the provider wins its bet. Because the seed is not needed to cover the NO winners, its 1,355.56 USDC profit comes out of funds that would otherwise contribute to the house take, which falls from 3,631.66 to 2,276.09 USDC.

A bet larger than the 1,369.94 shortfall is no longer free: at 2,000 USDC, the provider loses 2,000 instead of 1,369.94 if YES wins.

As a result, a seed provider can repeatedly extract house revenue from qualifying markets while taking no additional risk. It only needs to trade from another address and place the bet shortly before the market closes.

Remediation:

Consider charging a fee on every stake at buy time that accrues to the house in every outcome, so a hedge against the seed is never free.

Proof-of-Concept:

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity 0.8.27;

import { Test, console } from "forge-std/Test.sol";
import { MarketParlay } from "../src/MarketParlay.sol";
import { MarketScalar } from "../src/MarketScalar.sol";
import { MarketBinary } from "../src/MarketBinary.sol";
import { IMarketCategorical, IMarketCore } from "../src/IMarket.sol";
import { InitParams, Shape } from "../src/MarketTypes.sol";
import { MarketFactory } from "../src/MarketFactory.sol";
import { VaultRegistry } from "../src/VaultRegistry.sol";
import { Resolver } from "../src/Resolver.sol";
import { MockUSDC } from "./mocks/MockUSDC.sol";

contract SeedProviderHedgeTest is Test {
    uint256 internal constant USDC = 1e6;
    uint256 internal constant ALPHA_BPS = 300;
    uint256 internal constant B_MIN = 10_000 * USDC;
    uint256 internal constant CREATOR_BPS = 1000;
    uint256 internal constant YES = 0;
    uint256 internal constant NO = 1;

    /// @dev Values in the issue are rounded to cents.
    uint256 internal constant CENT = USDC / 100;

    MockUSDC internal usdc;
    MarketFactory internal factory;
    Resolver internal resolver;

    address internal owner = address(0xA11CE);
    address internal treasury = address(0x7E3A);
    address internal provider = address(0xC0DE); // self-funded creator = seed provider
    address internal providerAlt = address(0xC0DE2); // provider's second address
    address internal bob = address(0xB0B);
    address internal carol = address(0xCA201);

    struct Result {
        uint256 bobShares;
        uint256 carolShares;
```

```
    uint256 providerShares;
    uint256 seedRefund;
    uint256 houseTake;
    uint256 providerPayout;
    int256 providerPnl;
}

function setUp() public {
    usdc = new MockUSDC("USD Coin", "USDC", 6);
    VaultRegistry registry = new VaultRegistry(owner);
    factory = new MarketFactory(
        address(new MarketBinary()),
        address(new MarketScalar()),
        address(new MarketParlay()),
        address(registry),
        address(0),
        1_000_000
    );
    resolver = new Resolver(owner);

    usdc.mint(provider, 1_000_000 * USDC);
    usdc.mint(providerAlt, 1_000_000 * USDC);
    usdc.mint(bob, 1_000_000 * USDC);
    usdc.mint(carol, 1_000_000 * USDC);
}

function _market() internal returns (MarketBinary m, uint256 seed) {
    InitParams memory p = InitParams({
        collateral: address(usdc),
        resolver: address(resolver),
        alphaBps: ALPHA_BPS,
        bMin: B_MIN,
        treasury: treasury,
        creator: provider,
        creatorBps: CREATOR_BPS,
        closeTime: block.timestamp + 1 days,
        nOutcomes: 2,
        scalarLo: 0,
        scalarHi: 0,
        yieldVault: address(0),
        yieldHandlingBps: 0,
        legSizes: new uint256[](0),
```

```
            legLo: new uint256[](0),
            legHi: new uint256[](0),
            halfTaper: 0,
            allowlist: address(0)
        });
        uint256 before = usdc.balanceOf(provider);
        vm.startPrank(provider);
        usdc.approve(address(factory), type(uint256).max);
        m = MarketBinary(factory.createMarket(Shape.Binary, p));
        vm.stopPrank();
        seed = before - usdc.balanceOf(provider);
    }

    function _buy(MarketBinary m, address who, uint256 outcome, uint256 stake) internal returns (uint256
shares) {
        vm.startPrank(who);
        usdc.approve(address(m), stake);
        shares = m.buy(outcome, stake, 0, address(0));
        vm.stopPrank();
    }

    /// @dev Runs one full scenario on a fresh market.
    function _run(uint256 providerBet, uint256 winner) internal returns (Result memory r) {
        (MarketBinary m, uint256 seed) = _market();
        assertEq(seed, B_MIN, "binary seed = bMin");

        r.bobShares = _buy(m, bob, YES, 13_000 * USDC);
        r.carolShares = _buy(m, carol, NO, 4_000 * USDC);
        if (providerBet > 0) {
            r.providerShares = _buy(m, providerAlt, NO, providerBet);
        }

        vm.startPrank(owner);
        resolver.report(IMarketCategorical(address(m)), winner);
        r.houseTake = resolver.sweepMarket(IMarketCore(address(m))); // swept, after the seed refund
        vm.stopPrank();

        r.seedRefund = m.seedRefunded();
        if (providerBet > 0 && winner == NO) {
            vm.prank(providerAlt);
            r.providerPayout = m.claim();
        }
```

```
    // Provider P&L as in the issue: seed result plus bet result
    // (excluding its creatorBps share of the house take).
    r.providerPnl = int256(r.seedRefund) - int256(seed) + int256(r.providerPayout) - int256(providerBet);
}

function _log(string memory label, Result memory r) internal pure {
    console.log(label);
    console.log("  seed refund      ", r.seedRefund);
    console.log("  house take       ", r.houseTake);
    console.log("  provider payout  ", r.providerPayout);
    console.logInt(r.providerPnl);
}

function test_Issue13_SeedProviderHedgeIsFree() public {
    Result memory yesBase = _run(0, YES);
    Result memory noBase = _run(0, NO);
    Result memory yesHedge = _run(1_000 * USDC, YES);
    Result memory noHedge = _run(1_000 * USDC, NO);

    _log("YES wins, no provider bet", yesBase);
    _log("NO wins,  no provider bet", noBase);
    _log("YES wins, provider bets 1,000 NO", yesHedge);
    _log("NO wins,  provider bets 1,000 NO", noHedge);

    // Book state from the issue.
    console.log("Bob YES shares              ", yesBase.bobShares);
    console.log("Carol NO shares            ", yesBase.carolShares);
    console.log("provider NO shares          ", yesHedge.providerShares);

    // Table in the issue.
    console.log("YES, no bet: seed refund     ", yesBase.seedRefund);
    console.log("YES, bet: seed refund        ", yesHedge.seedRefund);
    console.log("YES, no bet: provider P&L    ", yesBase.providerPnl);
    console.log("YES, bet: provider P&L       ", yesHedge.providerPnl);
    console.log("NO, bet: provider payout     ", noHedge.providerPayout);
    console.log("NO, no bet: provider P&L     ", noBase.providerPnl);
    console.log("NO, bet: provider P&L        ", noHedge.providerPnl);
    console.log("NO, no bet: house take       ", noBase.houseTake);
    console.log("NO, bet: house take          ", noHedge.houseTake);

    // The core claims: the bet is free if YES wins, and the gain if NO wins
```

```
        // comes out of the house take.
        assertEq(yesHedge.providerPnl, yesBase.providerPnl, "hedge costs nothing when YES wins");
        assertApproxEqAbs(
            noBase.houseTake - noHedge.houseTake, uint256(noHedge.providerPnl), CENT, "gain is taken from
the house"
        );
    }

    /// @dev A bet larger than the 1,369.94 shortfall is no longer free.
    function test_Issue13_HedgeAboveShortfallIsNotFree() public {
        Result memory yesBase = _run(0, YES);
        Result memory yesBig = _run(2_000 * USDC, YES);
        assertApproxEqAbs(yesBig.providerPnl, -2_000_000_000, CENT, "loses the full 2,000");
        assertLt(yesBig.providerPnl, yesBase.providerPnl, "costs more than the shortfall alone");
    }
}
```

Commentary from the client:

“Partially fixed by incurring a cost for the hedge.”

FERM1-7 | Yield is shared by stake size only, so a late stake can take most of the accrued yield risk-free

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

contracts/src/MarketCore.sol#L1259-L1269

Description:

_yieldShare() splits the whole net yield pool by stake alone:

```
function _yieldShare(
    uint256 stake
) private view returns (uint256) {
    if (netYieldPool == 0 || totalVolume == 0) {
        return 0;
    }
    return Math.mulDiv(netYieldPool, stake, totalVolume);
}
```

However, it ignores when the stake entered and how long it was in the vault. A stake placed one minute before settlement, possibly never deposited, receives the same share as one that was in the vault for the whole market. Since **depositToYield()** also deposits the seed, which is not part of **totalVolume**, the seed's yield further enlarges the pool that late stakes can claim.

As a result, anyone can enter shortly before settlement and take most of the yield accrued by earlier bettors.

For example, when a market is voided, every stake is refunded at par. A late bettor therefore gets their entire stake back plus a stake-weighted share of the yield, even though their capital was never deposited into the vault, or was deposited only for a short time. This is especially easy on a conditional market: once its condition has publicly failed but betting is still open, the void is certain, so the late stake carries no risk at all.

Remediation:

Consider distributing yield based on time-weighted deposited capital (e.g. tracking stake-seconds since the last deposit), or only giving yield to stakes that were actually deposited into the vault.

Commentary from the client:

“The yield split may incorrect when the vault loses value and then recovers but it’s intended as joining later to a market should not have a different benefit in terms of vault losses. We don’t want vault fluctuations to dictate good or bad timing to enter a market.”

FERM1-11 | Base's Transaction Gas Cap Prevents Ordinary Buys in 32-Outcome Markets

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

contracts/src/MarketCore.sol:buy#L578-L596
contracts/src/MarketCore.sol:_credit#L659-L704
contracts/src/LsLmsr.sol:sharesBoughtImpl#L137-L167

Description:

The **buy** function lets a bettor purchase shares in one market outcome. It calls **_credit**, which uses **LsLmsr.sharesBought** to calculate the shares through repeated full-book cost calculations. This applies to every bettor, including users of the gasless **buyWithAuthorization** route.

All supported market configurations should be tradable on Base at normal stake sizes. In particular, **MAX_OUTCOMES** allows 32 outcomes so a five-leg binary parlay can be created, and the gas benchmark treats a 100 USDC buy in this configuration as supported.

However, the limit was chosen against Base's 100M block gas limit instead of its 16,777,216 per-transaction gas cap. The repository's own optimized benchmark shows that the 32-outcome, 100 USDC buy uses more than 25M gas. A user cannot raise the transaction gas limit to compensate: Base rejects a regular transaction above the cap, while a transaction at or below it runs out of gas. The authorized and basket-buy paths do at least the same pricing work and therefore do not provide a workaround.

An ordinary failure scenario is:

1. A creator deploys a five-leg binary parlay with 32 outcomes, which passes the **MAX_OUTCOMES** check.
2. A bettor submits a direct 100 USDC buy.
3. The buy needs more than 25M gas, so it cannot execute under Base's 16,777,216 transaction cap and reverts.
4. The market remains deployed, but its advertised trading flow is unavailable for this normal stake size.

Remediation:

Benchmark every advertised buy path against Base's per-transaction cap with enough safety margin, then lower **MAX_OUTCOMES** and constrain other gas-sensitive inputs as needed. The pricing implementation should also avoid repeating full-book calculations during bisection.

FERM1-9 | Protocol resolver fee ignores collateral decimals

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

contracts/src/MarketFactory.sol

Description:

The protocol resolver fee is calculated in raw token units without accounting for the collateral's decimals. Since **ResolverFeeConfig** uses one global **baseFee** and **perSecondRate**, the fee is only correctly priced for one decimal width.

For example, with **baseFee = 250,000** and **perSecondRate = 4**, a 90-day market pays:

- USDC (6 decimals): ~$31.35
- DAI (18 decimals): ~$0.000000000031

Thus, an 18-decimal stablecoin can effectively bypass the resolver fee.

Root Cause

The fee is taken directly from the global configuration:

```
terms.baseFee = feeConfig.baseFee();
terms.perSecondRate = feeConfig.perSecondRate();
terms.amount = terms.baseFee + terms.perSecondRate * horizon;
```

Unlike **minBMin** and **halfTaper**, these values are never scaled using **collateral.decimals()**.

Remediation:

Scale **baseFee**, **perSecondRate** according to the collateral's decimals, using the same approach as **minBMin** and **halfTaper**.

FERM1-16 | Parlay baskets add too much liquidity depth: each selected cell is counted, though only one can pay

Fixed

Severity:	Probability:	Impact:
Informational	Likely	Informational

Path:

contracts/src/MarketCore.sol#L1390-L1396

Description:

Parlay baskets pay an additional spread for every selected outcome cell, even though only one cell normally determines the payout. At the default **α = 300 bps**, this can cause users to lose money even when their bets win.

For example:
the market covers four games (Chiefs, Eagles, 49ers, Cowboys), creating 16 possible outcomes.

Alice bets $100 that the Chiefs win, regardless of the other three games. This covers 8 of the 16 possible outcomes. The market displays an 81.71% probability, suggesting a payout of $119.95 if she wins.

However, the addition spread increases her actual cost per share to $1.526, even though each winning share pays only $1. Her $100 buys 65.28 shares per outcome.

All four teams win, so Alice is correct. After the 1.99% settlement fee, she receives just $63.98, losing $36.02 on a winning bet.

The other bettors' payouts remain unchanged at $2,501.06, while the house take increases by $36.02.

Root cause

Liquidity depth is calculated from the total shares across all outcome cells:

```
b = max(bMin, α * totalShares); // LsLmsr.sol:54-65
```

Once α, rather than the seed floor **bMin**, determines the depth, every share added to any cell increases the total shares used to calculate liquidity depth. This introduces an additional spread of **α · H(p)** per share, where **H(p)** is the entropy of the outcome probabilities. However, the displayed prices (**priceBps** and **marginalPriceBps**) do not include this spread.

When buying a basket, it adds the same number of shares to every selected cell and recalculates the cost:

```
// LsLmsr.sol:212-231
for (uint256 i = 0; i < n; ++i) {
    prev[i] = q[cells[i]];
    q[cells[i]] = prev[i] + delta;
}

uint256 c = costImpl(q, alphaBps, bMin);
```

A basket covering |S| cells therefore increases **totalShares** by |S| · **delta**. When α determines the depth, this increases the liquidity depth used to price the purchase. However, only one selected cell normally determines the payout, so the basket receives at most one dollar per share when it wins.

The basket pays the additional spread for every selected cell, even though it can only receive one winning payout.

As the impact, users pay more than the displayed prices suggest and can lose money even when their bets win. The larger the basket, the more times the additional spread is charged. The excess accrues to the house rather than other bettors.

Remediation:

Calculate liquidity depth based on the shares that can actually be paid out, counting a basket's **delta** once rather than once per selected cell. Alternatively, scale and enforce **alpha** according to the number of outcome cells. Update displayed prices and documentation to disclose the actual basket cost.

Proof-of-Concept:

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity 0.8.27;

import { Test, console2 } from "forge-std/Test.sol";
import { MarketParlay } from "../../src/MarketParlay.sol";
import { MarketScalar } from "../../src/MarketScalar.sol";
import { MarketBinary } from "../../src/MarketBinary.sol";
import { InitParams, Shape } from "../../src/MarketTypes.sol";
import { MarketFactory } from "../../src/MarketFactory.sol";
import { VaultRegistry } from "../../src/VaultRegistry.sol";
import { FeeCurve } from "../../src/FeeCurve.sol";
import { LsLmsr } from "../../src/LsLmsr.sol";
import { MockUSDC } from "../mocks/MockUSDC.sol";

/// @title PoC: a correct parlay basket bet loses money
/// @notice 4-leg parlay (Chiefs, Eagles, 49ers, Cowboys; 16 outcomes), alpha = 300 bps (the
///         documented default), $100 seed. Ten ordinary tickets ($3,280) are placed. Alice then
///         bets $100 that the Chiefs win, whatever the other three games do: a basket over the 8
///         outcomes in which the Chiefs win. The Chiefs win and Alice is paid $63.98.
/// @dev Run: `forge test --match-path test/audit/PoC_CorrectParlayBasketLosesMoney.t.sol -vv`
contract PoC_CorrectParlayBasketLosesMoney is Test {
    uint256 internal constant USD = 1e6;
    uint256 internal constant W = 0; // the team wins
    uint256 internal constant L = 1; // the team loses

    MockUSDC internal usdc;
    MarketFactory internal factory;
    address internal creator = makeAddr("creator");
    address internal treasury = makeAddr("treasury");
    address internal alice = makeAddr("alice");
    address[10] internal crowd;

    function setUp() public {
        usdc = new MockUSDC("USD Coin", "USDC", 6);
        factory = new MarketFactory(
            address(new MarketBinary()),
            address(new MarketScalar()),
            address(new MarketParlay()),
            address(new VaultRegistry(address(this))),
```

```
        address(0),
        1_000_000 // production $1 opening floor
    );
    usdc.mint(creator, 1_000 * USD);
    vm.prank(creator);
    usdc.approve(address(factory), type(uint256).max);
}

function test_CorrectMarginalBasketLosesMoney() public {
    // ---------------------------------------- 1. the market and the crowd
    MarketParlay p = _createParlay();
    _placeCrowdTickets(p);
    uint256 snapNoAlice = vm.snapshotState(); // the same market without Alice's bet

    // ---------------------------------------- 2. Alice's bet: "Chiefs win"
    uint256[] memory chiefsWin = _cellsWhereChiefsWin(); // 8 of the 16 outcomes
    uint256 shownBps = p.marginalPriceBps(chiefsWin);

    // Cost of ONE basket share right now, straight from the cost function every buy uses
    // (no price impact): displayed probability + additional spread.
    uint256 marginalBps = _oneShareBasketCostBps(p, chiefsWin);

    usdc.mint(alice, 100 * USD);
    vm.startPrank(alice);
    usdc.approve(address(p), 100 * USD);
    uint256 sharesEach = p.buyBasket(chiefsWin, 100 * USD, 0, address(0));
    vm.stopPrank();
    uint256 avgCostPerShare = 100 * USD * 1e4 / sharesEach; // 1e-4 USD

    // ---------------------------------------- 3. resolve: Chiefs W, Eagles W, 49ers W, Cowboys W
    uint256 snapWithAlice = vm.snapshotState();
    p.resolveJoint(_v(W, W, W, W));
    uint256 keep = FeeCurve.keepBps(p.totalVolume(), p.halfTaper());
    uint256 alicePaid = p.claimable(alice);
    uint256 crowdPaidWith = _crowdClaimable(p);
    uint256 houseWith = p.houseTake();

    // Alice is paid the same in every one of the 8 results in which the Chiefs win.
    for (uint256 c = 0; c < 8; ++c) {
        vm.revertToState(snapWithAlice);
        snapWithAlice = vm.snapshotState();
        p.resolveJoint(_v(W, (c >> 2) & 1, (c >> 1) & 1, c & 1));
```

```
    assertEq(p.claimable(alice), alicePaid, "same payout in every Chiefs-win result");
}

// ---------------------------------------------- 4. the same result without Alice
vm.revertToState(snapNoAlice);
p.resolveJoint(_v(W, W, W, W));
uint256 crowdPaidWithout = _crowdClaimable(p);
uint256 houseWithout = p.houseTake();

// ---------------------------------------------- report
uint256 implied = 100 * USD * 10_000 / shownBps * keep / 10_000;
uint256 spreadBps = marginalBps - shownBps;

console2.log("");
console2.log("=== Context ===");
console2.log("4-leg parlay (Chiefs, Eagles, 49ers, Cowboys) = 16 possible outcomes");
console2.log("alpha 3% (documented default), creator seed $100.00");
console2.log("10 other bettors hold tickets worth $3,280.00 in total");

console2.log("");
console2.log("=== Step 1: what the screen shows ===");
console2.log("Alice bets $100.00 that the Chiefs win (8 of the 16 outcomes)");
_line("Displayed probability of 'Chiefs win'", _pct(shownBps));
_line("Payout the displayed price implies if right", _usd(implied));

console2.log("");
console2.log("=== Step 2: what one share really costs ===");
_line("Displayed price per share", _usd4(shownBps * 100));
_line("Additional spread per share (8 outcomes)", _usd4(spreadBps * 100));
_line("Real cost of one share", _usd4(marginalBps * 100));
_line("Most a winning share can pay", "$1.0000");

console2.log("");
console2.log("=== Step 3: what Alice receives for $100.00 ===");
_line("Shares in each of her 8 outcomes", _shares(sharesEach));
_line("Average price she paid per share", _usd4(avgCostPerShare * 100));

console2.log("");
console2.log("=== Step 4: the result ===");
console2.log("Chiefs W, Eagles W, 49ers W, Cowboys W  -> Alice is RIGHT");
console2.log("Only her shares in 'W W W W' pay; her other 7 outcomes pay $0");
```

```
        console2.log("");
        console2.log("=== Step 5: Alice's payout ===");
        _line("Settlement fee", _pct(10_000 - keep));
        _line(
            string.concat("Payout = ", _shares(sharesEach), " shares x $1 x ", _pct(keep)),
            _usd(alicePaid)
        );
        _line("Alice's profit on a CORRECT $100.00 bet", string.concat("-", _usd(100 * USD - alicePaid)));
        console2.log("(identical payout in all 8 results where the Chiefs win)");

        console2.log("");
        console2.log("=== Step 6: where her loss goes (same result, with vs without Alice) ===");
        _line("Other bettors are paid, without Alice", _usd(crowdPaidWithout));
        _line("Other bettors are paid, with Alice", _usd(crowdPaidWith));
        _line("House take, without Alice", _usd(houseWithout));
        _line("House take, with Alice", _usd(houseWith));
        _line("House take increase (= Alice's loss)", _usd(houseWith - houseWithout));
        console2.log("");

        // ---------------------------------------------- assertions
        assertEq(shownBps, 8171, "displayed 81.71%");
        assertEq(sharesEach / 1e4, 6528, "65.28 shares per outcome");
        assertGt(avgCostPerShare, 10_000, "a share costs more than the $1 it can ever pay");
        assertEq(alicePaid, sharesEach * keep / 10_000, "payout = shares x keep");
        assertLt(alicePaid, 100 * USD, "a correct bet returns less than its stake");
        assertEq(crowdPaidWith, crowdPaidWithout, "other bettors are paid exactly the same");
        assertApproxEqAbs(houseWith - houseWithout, 100 * USD - alicePaid, 1, "her loss goes to the house");
    }

    // ------------------------------------------------------ setup helpers

    /// 4 yes/no legs = 16 outcomes. alpha 300 bps, bMin $25 -> seed = $25 x log2(16) = $100.
    function _createParlay() internal returns (MarketParlay) {
        uint256[] memory legSizes = new uint256[](4);
        for (uint256 i = 0; i < 4; ++i) legSizes[i] = 2;
        InitParams memory prm = InitParams({
            collateral: address(usdc),
            resolver: address(this),
            alphaBps: 300,
            bMin: 25 * USD,
            treasury: treasury,
```

```
        creator: creator,
        creatorBps: 1000,
        closeTime: block.timestamp + 7 days,
        nOutcomes: 16,
        scalarLo: 0,
        scalarHi: 0,
        yieldVault: address(0),
        yieldHandlingBps: 0,
        legSizes: legSizes,
        legLo: new uint256[](0),
        legHi: new uint256[](0),
        halfTaper: 0,
        allowlist: address(0)
    });
    vm.prank(creator);
    return MarketParlay(factory.createMarket(Shape.Parlay, prm));
}

/// Ten ordinary 4-leg tickets, $3,280 in total, placed over five rounds.
function _placeCrowdTickets(
    MarketParlay p
) internal {
    uint256[10] memory cells = [
        _cell(W, W, W, W), _cell(W, W, W, L), _cell(W, W, L, W), _cell(W, L, W, W), _cell(L, W, W, W),
        _cell(W, W, L, L), _cell(L, L, W, W), _cell(W, L, L, W), _cell(L, W, L, W), _cell(L, L, L, L)
    ];
    uint256[10] memory stakes = [uint256(1200), 600, 400, 350, 300, 150, 100, 80, 60, 40];
    for (uint256 i = 0; i < 10; ++i) {
        crowd[i] = makeAddr(string.concat("user", vm.toString(i)));
        usdc.mint(crowd[i], stakes[i] * USD);
        vm.prank(crowd[i]);
        usdc.approve(address(p), type(uint256).max);
    }
    for (uint256 r = 0; r < 5; ++r) {
        for (uint256 i = 0; i < 10; ++i) {
            vm.prank(crowd[i]);
            p.buy(cells[i], stakes[i] * USD / 5, 0, address(0));
        }
    }
}
```

```
// ---------------------------------------------------------- measurement helpers

/// Cost (bps of $1) of one more share on every cell of the basket, from `LsLmsr.cost`.
function _oneShareBasketCostBps(
    MarketParlay p,
    uint256[] memory cells
) internal view returns (uint256) {
    uint256[] memory q = p.inventory();
    uint256 a = p.alphaBps();
    uint256 bm = p.bMin();
    uint256 base = LsLmsr.cost(q, a, bm);
    for (uint256 i = 0; i < cells.length; ++i) q[cells[i]] += 1e6; // +1 share each
    return (LsLmsr.cost(q, a, bm) - base) * 10_000 / 1e6;
}

function _crowdClaimable(
    MarketParlay p
) internal view returns (uint256 sum) {
    for (uint256 i = 0; i < 10; ++i) sum += p.claimable(crowd[i]);
}

function _cellsWhereChiefsWin() internal pure returns (uint256[] memory cells) {
    cells = new uint256[](8);
    for (uint256 i = 0; i < 8; ++i) cells[i] = i; // leg 1 (Chiefs) is the leading digit
}

function _cell(
    uint256 a,
    uint256 b,
    uint256 c,
    uint256 d
) internal pure returns (uint256) {
    return ((a * 2 + b) * 2 + c) * 2 + d;
}

function _v(
    uint256 a,
    uint256 b,
    uint256 c,
    uint256 d
) internal pure returns (uint256[] memory v) {
```

```
        v = new uint256[](4);
        v[0] = a;
        v[1] = b;
        v[2] = c;
        v[3] = d;
    }

    // ---------------------------------------------------------- formatting helpers

    /// "label ..... value", aligned.
    function _line(
        string memory label,
        string memory value
    ) internal pure {
        bytes memory pad = new bytes(bytes(label).length < 48 ? 48 - bytes(label).length : 1);
        for (uint256 i = 0; i < pad.length; ++i) pad[i] = " ";
        console2.log(string.concat(label, ":", string(pad), value));
    }

    /// 6-decimal USDC amount -> "$1,234.56" (rounded to the cent).
    function _usd(
        uint256 a
    ) internal view returns (string memory) {
        uint256 cents = (a + 5_000) / 10_000;
        return string.concat("$", _commas(cents / 100), ".", _two(cents % 100));
    }

    /// 6-decimal USDC amount -> "$1.5318" (rounded to 1/10,000 of a dollar).
    function _usd4(
        uint256 a
    ) internal view returns (string memory) {
        uint256 v = (a + 50) / 100;
        string memory f = vm.toString(v % 10_000);
        while (bytes(f).length < 4) f = string.concat("0", f);
        return string.concat("$", _commas(v / 10_000), ".", f);
    }

    /// basis points -> "81.71%".
    function _pct(
        uint256 bps
    ) internal view returns (string memory) {
        return string.concat(vm.toString(bps / 100), ".", _two(bps % 100), "%");
```

```
    }

    /// 6-decimal share amount -> "65.28".
    function _shares(
        uint256 s
    ) internal view returns (string memory) {
        uint256 h = (s + 5_000) / 10_000;
        return string.concat(_commas(h / 100), ".", _two(h % 100));
    }

    function _two(
        uint256 x
    ) internal view returns (string memory) {
        return x < 10 ? string.concat("0", vm.toString(x)) : vm.toString(x);
    }

    function _commas(
        uint256 x
    ) internal view returns (string memory out) {
        if (x < 1000) return vm.toString(x);
        string memory rest = vm.toString(x % 1000);
        while (bytes(rest).length < 3) rest = string.concat("0", rest);
        return string.concat(_commas(x / 1000), ",", rest);
    }
}
```

Output:

```
=== Context ===
4-leg parlay (Chiefs, Eagles, 49ers, Cowboys) = 16 possible outcomes
alpha 3% (documented default), creator seed $100.00
10 other bettors hold tickets worth $3,280.00 in total

=== Step 1: what the screen shows ===
Alice bets $100.00 that the Chiefs win (8 of the 16 outcomes)
Displayed probability of 'Chiefs win':          81.71%
Payout the displayed price implies if right:    $119.95

=== Step 2: what one share really costs ===
Displayed price per share:                  $0.8171
Additional spread per share (8 outcomes):       $0.7089
Real cost of one share:                     $1.5260
Most a winning share can pay:                $1.0000

=== Step 3: what Alice receives for $100.00 ===
Shares in each of her 8 outcomes:             65.28
Average price she paid per share:             $1.5318

=== result ===
Chiefs W, Eagles W, 49ers W, Cowboys W  -> Alice is RIGHT
Only her shares in 'W W W W' pay; her other 7 outcomes pay $0

=== Alice's payout ===
Settlement fee:                          1.99%
Payout = 65.28 shares x $1 x 98.01%:          $63.98
Alice's profit on a CORRECT $100.00 bet:      -$36.02
(identical payout in all 8 results where the Chiefs win)

=== where her loss goes (same result, with vs without Alice) ===
Other bettors are paid, without Alice:        $2,501.06
Other bettors are paid, with Alice:           $2,501.06
House take, without Alice:                  $778.94
House take, with Alice:                     $814.96
House take increase (= Alice's loss):         $36.02
```

FERM1-2 | priceBps() is documented as the marginal price but returns the softmax probability

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

docs/src/mechanism.md
contracts/src/LsLmsr.sol#L111-L116

Description:

priceBps() and the **priceBps** in the **Bought** event return the market's displayed probability. However, the documentation defines the marginal price as the derivative of the cost function, i.e. the price a user pays for the next share:

> The marginal price of outcome `i` is `∂C/∂qᵢ`, always a probability in `(0, 1)`,
> and the `N` prices sum to one; ...

And the NatSpec describes **priceBps()** as exactly this marginal price:

```
/// @notice Marginal price of `outcome` in basis points:
///         `2^(q_i/b) / Sigma_j 2^(q_j/b)`, from the shifted exponentials so
///         it stays in range. Each price is a true probability and the N
///         prices sum to 10_000 (up to flooring dust). ...
function priceBpsImpl(
```

```
/// @notice Marginal price of `outcome` in basis points.
function priceBps(
```

When the market's liquidity grows beyond **bMin**, the actual cost of the next share becomes higher than the displayed probability.

The displayed value is the softmax probability **p_i**, which equals the marginal cost only while **b** is fixed. Once **alphaBps * Σq > bMin**, **b = α * Σq** also grows with every share bought, so the cost function **C = b * log2(Σ_j 2^(q_j / b))** gives:

```
∂C/∂q_i = (∂b/∂q_i) * log2(Z) + b * ∂log2(Z)/∂q_i        Z = Σ_j 2^(q_j / b)
        = α * log2(Z) + (p_i − α * Σ_j p_j * q_j / b)
        = p_i + α * H(p)                                  H(p) = −Σ_j p_j * log2(p_j)
```

So the actual marginal cost is higher than **priceBps()** by **α * H(p)**.

The trade itself is calculated correctly by **buy()** using **sharesBought()**. The issue is that an integrator relying on the NatSpec and documentation would use **priceBps()** as the marginal price to calculate the expected shares and **minSharesOut**.

Remediation:

Clarify that **priceBps()** represents the displayed probability, not the actual execution price. Integrators should use **sharesBought()** or a dedicated quote function to calculate the expected number of shares.

FERM1-3 | Seeded parlays refund losing bets when the winning cell is empty

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Description:

On a seeded market, the seed acts as the counterparty of the bettors: when the winners are owed more than the pot, the seed pays the difference. So the seed loses whenever a bettor wins against a thin book.

However, when a market resolves to a cell that nobody bought, **MarketCore** treats it as void and refunds all stakes:

```
function _isVoid() internal view returns (bool) {
    return _weightedShares(_q) == 0;
}
```

In that case every losing stake is returned, so the seed provider, which pays the bettor when the bettor wins, gains nothing when the bettor loses.

In a parlay with a lot of outcomes but thin liquidity (16 cells for 4 legs, 32 for 5 legs), many cells end up with no shares. Consider a bettor who buys a single cell. A single cell rarely wins, but at resolution there are three cases:

Result	Bettor	Seed
The bettor's cell wins	paid out; on a thin book most of the payout comes from the seed	loses
Another cell held by someone else wins	loses the stake	-
An empty cell wins	refunded in full (void)	-

The second case is the only one in which the bettor loses, and it requires someone else to hold the exact realized cell. When the bettor is alone, or when bets are concentrated on a few cells, it rarely or never happens. The bettor therefore keeps the upside of the bet, which the seed pays for, while its downside is mostly removed by the void.

As a result, a bettor can repeatedly take a positive-expectation position against the seed in a thin liquidity market.

Remediation:

Consider charging a fee on each bet at execution time that accrues to the seed provider and is not refunded on a void, so the seed is compensated for the risk it takes as the counterparty regardless of how the market resolves.

FERM1-5 | Seed-cover scaling can reduce an earlier winner's payout after a later bet

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/src/MarketCore.sol#L1067-L1068
docs/src/mechanism/evidence.md

Description:

Following the documentation, a later bettor should not be able to dilute an earlier bettor's position:

```
## No dilution: an early bet is untouchable

A separate scripted scenario isolates the core fairness promise. An early small
bettor's payout is measured twice: once in a quiet market, once with a late whale
piling in on the same side. Under LS-LMSR the early bettor's payout is
**byte-identical** in both runs — the whale cannot dilute shares already bought.
```

However, on a seeded market (**bMin > 0**), this does not hold when the amount owed to winners exceeds the pot. At resolution, **_setSeedScale()** compares the nominal winner redemption with **totalVolume**, and scales every winning claim down by up to 5% when the pot does not cover it:

```
function _setSeedScale() private {
    seedScaleBps = BPS_DENOM;
    uint256 nominal = Math.mulDiv(
        unclaimedWinnerShares, FeeCurve.keepBps(totalVolume, halfTaper), BPS_DENOM
    );
    if (nominal <= totalVolume || requiredSeed() == 0) {
        emit SeedScaleFixed(BPS_DENOM);
        return;
    }
    uint256 tight = (totalVolume * BPS_DENOM + (BPS_DENOM - 1)) / nominal;
    seedScaleBps = Math.max(tight, SEED_COVER_FLOOR_BPS);
    emit SeedScaleFixed(seedScaleBps);
}
```

The resulting **seedScaleBps** is applied uniformly to every winning claim through **_seedScaled()**, and it is computed only from the final state of the market. Since a later bet on the winning outcome can increase **unclaimedWinnerShares** (and therefore **nominal**) by more than it increases **totalVolume**, pushing **nominal** above the pot or lowering **tight**.

As a result, a later bet on the same side can reduce the payout of an earlier winner by up to 5%, even though the earlier winner's shares were fixed at purchase. The final payout of a position therefore cannot be determined from its purchase-time price, which contradicts the documented "no dilution" guarantee.

Remediation:

Consider documenting that winner payouts on seeded markets are subject to a seed-cover scale of up to 5% that depends on the final market state, or limiting the scale so that it cannot reduce payouts of shares bought before the bet that caused the shortfall.

FERM1-8 | Seed is deposited into the yield vault, contrary to the documentation

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/src/MarketCore.sol#L743
contracts/src/MarketCore.sol#L855
docs/src/mechanism/collateral.md

Description:

The documentation states that the seed is a static maker buffer and is not deposited into the yield vault:

> The seed is a **static maker buffer**: it is not deployed to the yield rail, so the [yield](#yield-on-idle-collateral) distribution and the seed refund are independent — only pooled stakes are routed to a vault, and the seed is refunded flat.

However, **depositToYield()** deposits the entire market balance, which includes the seed transferred by the factory at creation:

```
uint256 assets = collateral.balanceOf(address(this));
// Nothing idle to deploy (every buy may already be in the vault) — a
// benign no-op rather than a revert, so the operator can poll safely.
if (assets > 0) {
    principalDeployed += assets;
    collateral.forceApprove(address(yieldVault), assets);
    uint256 mintedShares = yieldVault.deposit(assets, address(this));
    ...
}
```

The implementation itself also assumes the opposite of the documentation. **_finalizeRecovery()** counts the seed as part of the deposited principal:

```
// The seed is maker capital deposited into the vault alongside the
// stakes, so it backs the winner obligation exactly like the pot does.
...
uint256 backing = totalVolume + requiredSeed();
```

As a result, the documented guarantees that only pooled stakes are routed to the vault, that the yield distribution and the seed refund are independent, and that the seed is refunded flat do not hold:

- The seed is exposed to vault losses. For example, a funder supplies a $10,000 seed and nobody bets. The operator calls **depositToYield()**, and the vault loses 10%. The funder receives only $9,000 back, despite no user ever betting.
- The seed's yield is not credited in proportion to the seed. The gross yield is split into **yieldHandlingFee**, which flows into the house take and reaches the creator only through its **creatorBps** share, and **netYieldPool**, which is distributed to bettors pro-rata to their stake. Bettors therefore receive most of the yield generated by the seed, and the seed provider's share does not scale with the capital it supplied.

Remediation:

Consider either excluding the seed from **depositToYield()** (e.g. depositing at most **balance - requiredSeed()**) so the implementation matches the documentation, or updating the documentation to state that the seed is deposited alongside the stakes, bears vault losses, and does not earn its share of the yield.

FERM1-10 | Sub-basis-point prices cap fills while charging the full stake

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/src/LsLmsr.sol:sharesBoughtImpl]#L117-166, L295-L317

Description:

sharesBoughtImpl converts a user's stake into shares by searching for the largest share amount whose LMSR cost is no greater than the stake. Regular and gasless buys reach it through **MarketCore._credit**, while parlay basket buys use the equivalent **basketSharesImpl** flow. These entry points are available to ordinary bettors.

The search should start with an upper bound that is known to be unaffordable. Bisection can then return the largest affordable amount, after which the market credits those shares and pulls the user's full stake.

However, **priceBpsImpl** rounds prices down to whole basis points. When an outcome's real price is positive but below one basis point, it returns zero. **bracket** then changes that zero to one basis point and derives the search limit from it:

```
uint256 priceBpsValue = Math.max(priceBpsImpl(outcome, q, alphaBps, bMin), 1);
uint256 hiBound = Math.mulDiv(target, BPS_DENOM, priceBpsValue) + 1;
```

Rounding the price up makes **hiBound** too small, so it is not necessarily an upper bound at all. **sharesBoughtImpl** never checks whether that bound is unaffordable and cannot find any larger fill. The basket path has the same issue when all selected cell prices round to zero. **MarketCore.buy** and **MarketParlay.buyBasket** still transfer and account for the full stake, with no refund for the underfill.

For example, consider a 6-decimal binary market with **alphaBps = 300**, **bMin = 1,000,000**, inventory **[1,000,000,000,000, 0]**, and a **1,000,000** stake on the second outcome:

1. The second outcome's real price is about **0.000000924** basis points, but **priceBpsImpl** returns zero.
2. bracket substitutes one basis point and caps the search near **10,000,000,000** share units.
3. A fill of **250,000,000,000** share units is still affordable and costs only about **51,590** units under **costImpl**, but bisection cannot reach it.
4. With a permissive minSharesOut, the market credits the capped fill and pulls the full 1,000,000 stake. The bettor receives far less payout exposure if the outcome wins, while the excess collateral remains in the market.

```
function sharesBoughtImpl(
    uint256 outcome,
    uint256 stake,
    uint256[] memory q,
    uint256 alphaBps,
    uint256 bMin
) internal pure returns (uint256) {
    if (stake == 0 || outcome >= q.length) {
        return 0;
    }
    uint256 base = costImpl(q, alphaBps, bMin);
    (uint256 lo, uint256 hi) = bracket(outcome, stake, q, alphaBps, bMin);
    for (uint256 i = 0; i < BISECT_STEPS; ++i) {
        if (hi - lo <= 1) {
            break;
        }
        uint256 mid = lo + (hi - lo) / 2;
        if (costIncrement(outcome, mid, base, q, alphaBps, bMin) <= stake) {
            lo = mid;
        } else {
            hi = mid;
        }
    }
    return lo;
}
```

Remediation:

Build the bracket from **costIncrement**: grow the upper bound until it is unaffordable, then run bisection. Apply the same approach to **basketBracket** and test that the returned fill is affordable while the next share is not.

FERM1-12 | Outstanding allowlist signature can override the owner's later direct update

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/src/Allowlist.sol#L90

Description:

The **Allowlist** owner can update bettors directly through **setAllowed()**, or gaslessly by signing a **SetAllowed** message that anyone can submit through **setAllowedWithSig()**:

```
function setAllowed(
    address[] calldata bettors,
    bool ok
) external onlyOwner {
    _setAllowed(bettors, ok);
}

function setAllowedWithSig(
    address[] calldata bettors,
    bool ok,
    uint256 nonce_,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
    if (nonce_ != nonce) {
        revert InvalidNonce();
    }
    ...
    nonce = nonce_ + 1;
    _setAllowed(bettors, ok);
}
```

However, the signed message has no deadline, and the direct **setAllowed()** does not advance **nonce**. As a result, a signed update that has not been submitted yet stays valid indefinitely, and the owner cannot invalidate it with a direct call.

However, the signed message has no deadline, and the direct **setAllowed()** does not advance **nonce**. As a result, a signed update that has not been submitted yet stays valid indefinitely, and the owner cannot invalidate it with a direct call.

For example:

1. The owner signs **SetAllowed([Alice], true, nonce = 1)** and hands it to the relayer.
2. Before it is submitted, the owner decides Alice must not bet anymore and calls **setAllowed([Alice], false)** directly. **nonce** is still **1**.
3. The relayer (or anyone holding the signature) submits the signed message. The nonce still matches, so Alice is allowed again and the owner's latest decision is silently reverted.

Note that the codebase already handles this case elsewhere: **CreatorResolver** and **MarketCore.sweep()** both consume the current nonce on the direct path, so an older outstanding signature is invalidated.

Remediation:

Consider advancing **nonce** in **setAllowed()** as well, and adding a **deadline** to the **SetAllowed** message:

```
function setAllowed(
    address[] calldata bettors,
    bool ok
) external onlyOwner {
++  nonce += 1;
    _setAllowed(bettors, ok);
}
```

FERM1-13 | Unused _isScalar() function

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/src/MarketCore.sol#L480-L482
contracts/src/MarketScalar.sol#L56-L58

Description:

MarketCore declares the internal virtual function **_isScalar()**, which **MarketScalar** overrides:

```
// MarketCore
function _isScalar() internal view virtual returns (bool) {
    return false;
}

// MarketScalar
function _isScalar() internal view override returns (bool) {
    return scalarHi > scalarLo;
}
```

However, the function is never called anywhere in the codebase. Its NatSpec states that it is "consumed by the shared settlement tail", but **_settle()** and the other settlement functions never reference it; each **resolve*()** entry point builds its own payout weights instead.

Remediation:

Consider removing **_isScalar()** from **MarketCore** and **MarketScalar**, or updating its NatSpec if it is meant for future use.

FERM1-14 | Dispute incentives weaken as the market grows

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/src/DisputeResolver.sol#L680

Description:

A proposal in the **DisputeResolver** can only be challenged by posting a bond, which scales linearly with the market's **totalVolume**:

```
function disputeBond(
    address market
) public view returns (uint256 bond) {
    ...
    uint256 pot = IMarketCore(market).totalVolume();
    return Math.max(
        cfg.minBond,
        Math.mulDiv(pot, cfg.bondBps, BPS_DENOM)
    );
}
```

The bond has no upper cap. With **bondBps = 100**, disputing a 10M USDC market requires a 100,000 USDC bond (which is a significant amount of money for a person).

However, the disputer has little reason to post it, even when the proposal is wrong:

1. A correct dispute only gets its bond back. In **arbitrate()**, an overturned proposal calls **_refundBond()** and nothing more, so there is no reward for the disputer.
2. A correct dispute can still lose the bond. If the arbiters do not rule within **arbiterTimeout**, **_closeWindow()** finalizes the original proposal and calls **_forfeitBond()** regardless of whether the dispute was right:

```
else if (st.status == Status.Disputed) {
    if (block.timestamp < st.disputedAt + cfg.arbiterTimeout) {
        revert NotArbiterTimedOut();
    }
    uint256 forfeited = st.bond;
```

```
        st.bond = 0;
        _finalize(market, st.proposed);
        _forfeitBond(market, forfeited);
    }
```

3. One disputer posts the whole bond, while the benefit of overturning a wrong outcome is shared by every holder of the correct outcome. Only a holder whose own position is worth more than the bond has a reason to dispute.

As a result, the larger the market, the more a wrong (or compromised) proposal is worth, and the less likely it is to be challenged.

Remediation:

Consider one or more of the following:

- Capping the bond, e.g. **min(max(minBond, bondBps * pot), maxBond)**.
- Rewarding a successful disputer, for example from a proposer bond or a share of the house take.
- Refunding the bond instead of forfeiting it when the arbiters time out.

Commentary from the client:

“The risk will be documented internally.”

FERM1-15 | Misplaced NatSpec for _credit()

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/src/MarketCore.sol#L639-L657

Description:

The NatSpec from line 639 - 644 describing **_credit()** is placed above **_requireBettorAllowed()** instead of above **_credit()** itself:

```
/// @dev The shared body of a buy, run before collateral is pulled: validates
///      the market is open and the outcome is in range, prices the stake,
///      enforces the slippage guard, records `shares` of `outcome` against
///      `buyer`'s position, advances the book and volume, and emits
///      {Bought}. The caller performs the collateral transfer last (CEI), so
///      a failed transfer reverts these writes.
/// @dev Enforces the optional betting allowlist. A zero `allowlist` is
///      permissionless and short-circuits before any external call ...
function _requireBettorAllowed(
    address buyer
) internal view {
    ...
}

function _credit(
    address buyer,
    uint256 outcome,
    uint256 stake,
    uint256 minSharesOut,
    address sponsor
) internal returns (uint256 shares) {
```

As a result, both **@dev** blocks are attached to **_requireBettorAllowed()**, which wrongly describes it as pricing the stake and recording shares, while **_credit()** has no documentation.

Remediation:

Consider moving the first **@dev** block directly **above _credit()**.

hexens x FERMAH