Myriad Prediction Market Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Immeas

Chinmay

InAllHonesty

July 25, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Myriad is a prediction market platform using an AMM to manage prices. It relies on Reality.eth as the oracle for market outcomes. Markets can be denominated in real tokens, like USDC, or in "fantasy" tokens abstracted by the `FantasyERC20` contract.

4.1 Actors and Roles

1. Actors

- **Protocol Owner**: Deploys and manages `PredictionMarketV3_4`, including adding manager contracts, pausing the protocol, performing emergency withdrawals, and upgrading the contract.
- **Market Manager**: Responsible for adding markets, pausing markets, changing market close dates, and force-resolving market outcomes.
- **Reality.eth**: An external oracle service that supplies market outcomes.
- **Liquidity Providers**: Supply liquidity to the prediction markets and earn fees. Similar to impermanent loss in AMMs, Myriad LPs are also partially exposed to market outcomes. In unbalanced markets, they effectively take the opposite position.
- **Users**: Trade and speculate on market outcomes.

2. Roles

- **owner**: Can upgrade `PredictionMarketV3_4`, pause the protocol, add or remove trusted managers, and perform emergency withdrawals of tokens from the contract.
- **manager**: Represented by the `PredictionMarketV3Manager` contract, which manages admins with permissions to force-resolve outcomes, adjust close dates, and pause specific markets.
- **Land Admin**: Oversees "lands" that group markets. Land admins can add or remove market managers.

4.2 Key Components

- **PredictionMarketV3_4**: Core prediction market contract responsible for buying/selling outcome tokens and managing liquidity.
- **PredictionMarketV3Manager**: Extends `LandFactory` and serves as the management layer for market admins in `PredictionMarketV3_4`.
- **FantasyERC20**: An ERC20 implementation used for "fantasy" tokens, which act as play-money assets for prediction markets.

4.3 Centralization Risks

The protocol is heavily controlled by the Myriad team, so using it requires a high degree of trust in Myriad's operational security. If privileged admin wallets were compromised, the consequences could be severe. Admin keys and related infrastructure must be secured with extreme caution.

5 Audit Scope

```
contracts/PredictionMarketV3_4.sol
contracts/LandFactory.sol
contracts/PredictionMarketV3Manager.sol
```

6 Executive Summary

Over the course of 8 days, the Cyfrin team conducted an audit on the Myriad Prediction Market smart contracts provided by Myriad. In this period, a total of 15 issues were found.

During the audit, 6 low severity issues were identified. These included edge cases involving invalid responses from Reality.eth, unreachable `closed` market state, unexpected behavior around `closesAtTimestamp`, the absence of slippage and deadline protections, and limitations when attempting to sell a full position.

In addition, 9 informational findings were reported. These covered areas such as code and test clarity, naming conventions, and general improvements to enhance maintainability and developer experience.

Summary

Project Name	Myriad Prediction Market
Repository	polkamarkets-js
Commit	24f1394be94d...
Fix Commit	dbcfdfabe130...
Audit Timeline	Jul 15th - Jul 24th, 2025
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	6
Informational	9
Gas Optimizations	0
Total Issues	15

Summary of Findings

[L-1] Ambiguous `-1` return value in `PredictionMarketV3_4::getMarketResolvedOutcome`	Resolved
[L-2] Markets Stuck in 'Open' State even if `block.timestamp > markets[marketId].closesAtTimestamp == true`	Resolved
[L-3] Operations done right at the `closesAtTimestamp`	Resolved
[L-4] Lack of slippage protection in liquidity functions	Resolved
[L-5] Consider adding a `deadline` parameter to user facing calls	Acknowledged
[L-6] Consider adding share-based sell function to avoid dust shares	Acknowledged
[I-1] Deprecated `testFail` Usage in Tests	Resolved
[I-2] Consider Using `Ownable2StepUpgradeable`	Resolved
[I-3] Consider enforcing a `minAmount` to prevent rounding exploits	Acknowledged
[I-4] Disallow single-outcome markets	Resolved
[I-5] Add explicit check to prevent underflow revert in `PredictionMarketV3_4::calcSellAmount`	Resolved
[I-6] Unused field `MarketResolution.resolved`	Resolved
[I-7] "Weight" and `poolWeight` serves different meanings throughout the code/documentation	Resolved
[I-8] Unnecessary transfer in `PredictionMarketV3_4::mintAndCreateMarket`	Resolved
[I-9] Missing event for important state changes	Resolved

7 Findings

7.1 Low Risk

7.1.1 Ambiguous `-1` return value in `PredictionMarketV3_4::getMarketResolvedOutcome`

Description: In `PredictionMarketV3_4::getMarketResolvedOutcome`, there is logic to indicate when a market resolution is still pending:

```
function getMarketResolvedOutcome(uint256 marketId) public view returns (int256) {
  Market storage market = markets[marketId];

  // returning -1 if market still not resolved
  if (market.state != MarketState.resolved) {
    return -1;
  }

  return int256(market.resolution.outcomeId);
}
```

The function returns `-1` if the market has not yet reached the `resolved` state. However, if the question is resolved via Reality.eth (Realitio), the returned outcome may be `0xffff...ffff` (`type(uint256).max`), which Reality.eth uses to signal an invalid answer, as noted in their documentation:

By convention all supported types use 0xff...ff (aka bytes32(type(uint256).max)) to mean “invalid”.

Because `int256(type(uint256).max)` equals `-1`, a resolved market with an invalid answer will also return `-1` from `getMarketResolvedOutcome()`. This creates ambiguity: services interpreting `-1` as “unresolved” could mistakenly treat an invalid resolution as if the market were still pending.

Impact: Consumers relying on `getMarketResolvedOutcome()` to return `-1` only when the market is unresolved may misinterpret the state. A market could be resolved with an invalid answer from Reality.eth, but still appear unresolved due to the reused sentinel value.

Recommended Mitigation: Consider returning a different sentinel value to represent unresolved state. Since the number of outcomes is limited to 32, any number greater than or equal to 32 could serve as a distinct "unresolved" indicator. However, avoid using `-2`, as it is another special value used by Reality.eth to indicate unresolved answer.

Myriad: Fixed in PR, commit `27ccc51`

Cyfrin: Verified. `-3` is now used as unresolved outcome.

7.1.2 Markets Stuck in 'Open' State even if `block.timestamp > markets[marketId].closesAtTimestamp == true`

Description: Functions `_buy`, `_sell`, `_addLiquidity` or `_removeLiquidity` use the `timeTransitions(marketId)` modifier followed by `atState(marketId, MarketState.open)`.

If someone tries to do one of these operations in this context: `block.timestamp > markets[marketId].closesAtTimestamp && markets[marketId].state == MarketState.open` then the code calls `_nextState(marketId)` which changes the state from `open` to `closed`: `market.state = MarketState(uint256(market.state) + 1)`. But when the next modifier runs atState(marketId, MarketState.open) everything reverts. The market state change doesn't persist, leaving the `markets[marketId].state = MarketState.open`.

Furthermore, the only time when `timeTransitions` successfully executes the `_nextState` transition, in that context, is inside `resolveMarketOutcome`. Making the `closed` state only a transitory step between `open` and `resolved` that takes place between the execution of `resolveMarketOutcome` function's modifiers.

Impact: Markets remain in `open` state even if `block.timestamp > markets[marketId].closesAtTimestamp == true` when they should be closed, possibly causing UX issues.

Any tx attempting to `_buy`, `_sell`, `_addLiquidity` or `_removeLiquidity` will get reverted by the `atState(marketId, MarketState.open)` modifier, wasting caller's gas.

Proof of Concept: Add the following test inside `PredictionMarket.t.sol`:

```
    function test_MarketStuckInOpenState() public {
        // Create market that closes in 1 hour
        uint32 closeTime = uint32(block.timestamp + 3600);

        PredictionMarketV3_4.CreateMarketDescription memory desc =
        ↪   PredictionMarketV3_4.CreateMarketDescription({
            value: VALUE,
            closesAt: closeTime,
            outcomes: 2,
            token: IERC20(address(tokenERC20)),
            distribution: new uint256[](0),
            question: "PoC Market State Issue",
            image: "test",
            arbitrator: address(0x1),
            buyFees: PredictionMarketV3_4.Fees({fee: 0, treasuryFee: 0, distributorFee: 0}),
            sellFees: PredictionMarketV3_4.Fees({fee: 0, treasuryFee: 0, distributorFee: 0}),
            treasury: treasury,
            distributor: distributor,
            realitioTimeout: 3600,
            manager: IPredictionMarketV3Manager(address(manager))
        });

        uint256 marketId = predictionMarket.createMarket(desc);

        // Verify market is initially open
        (PredictionMarketV3_4.MarketState state, uint256 closesAt,,,,) =
        ↪   predictionMarket.getMarketData(marketId);
        assertEq(uint256(state), uint256(PredictionMarketV3_4.MarketState.open));
        assertEq(closesAt, closeTime);

        console.log("BEFORE TIME WARP");
        console.log("Market state (0=open, 1=closed, 2=resolved):", uint256(state));
        console.log("Time until close:", closesAt - block.timestamp);

        // TIME WARP: Move 2 hours into the future (1 hour past close time)
        vm.warp(block.timestamp + 7200);

        console.log("AFTER TIME WARP");
        console.log("Market closes at:", closesAt);
        console.log("Time past close:", block.timestamp - closesAt);

        (state,,,,,) = predictionMarket.getMarketData(marketId);
        console.log("Market state (should be 1=closed, but shows):", uint256(state));

        // Prove the market is logically closed but state is wrong
        assertTrue(block.timestamp > closesAt, "We are past close time");
        assertEq(uint256(state), uint256(PredictionMarketV3_4.MarketState.open), "Market incorrectly
        ↪   shows as open!");

        // ATTEMPT TO TRADE: This should fail due to modifier ordering issue
        console.log("ATTEMPTING TRADE (should fail)");

        address trader = makeAddr("trader");
        deal(address(tokenERC20), trader, 1 ether);

        vm.startPrank(trader);
        tokenERC20.approve(address(predictionMarket), type(uint256).max);
        vm.expectRevert(bytes("!ms")); // Market state error
        predictionMarket.buy(marketId, 0, 0, 0.1 ether);
        vm.stopPrank();

        console.log("Trade failed as expected due to modifier");
    }
```

Recommended Mitigation: Consider removing the state `closed` and using the `closesAtDate` to indicate whether the market is open/closed, or make the transition work.

Myriad: Fixed in PR#85, commit `627d5be`

Cyfrin: Verified. `getMarketData` now calls a new function `getMarketState` which returns `closed` if `market.state == MarketState.open && block.timestamp >= market.closesAtTimestamp`.

7.1.3 Operations done right at the `closesAtTimestamp`

Description: The protocol checks the time validity of all user facing methods through the following modifier:

```
  modifier timeTransitions(uint256 marketId) {
    if (block.timestamp > markets[marketId].closesAtTimestamp && markets[marketId].state ==
    ↪  MarketState.open) {
      _nextState(marketId);
    }
    _;
  }
```

This condition `block.timestamp > markets[marketId].closesAtTimestamp` excludes the case where `block.timestamp = markets[marketId].closesAtTimestamp`, allowing buy, sell, addLiquidity and removeLiquidity operations right on market closure.

Impact: Potential risk free earnings depending on the market status and correct answer.

Proof of Concept: Add the following to `PredictionMarket.t.sol`

```
    function testTradeRightAtMarketClose() public {
        uint256 marketId = _createTestMarket();

        // Get market close time
        (,uint256 closesAt,,,,) = predictionMarket.getMarketData(marketId);

        // Set timestamp to exact close time
        vm.warp(closesAt);

        // This should fail - market transitions to closed due to timeTransitions modifier // NB it
        ↪  doesn't
        // vm.expectRevert(bytes("!ms")); // Market state error
        predictionMarket.buy(marketId, 0, 0, VALUE);
    }


    function testLiquidityOperationsAtMarketClose() public {
        uint256 marketId = _createTestMarket();

        // Get market close time
        (,uint256 closesAt,,,,) = predictionMarket.getMarketData(marketId);

        // Add more liquidity before close
        vm.warp(closesAt - 1);
        predictionMarket.addLiquidity(marketId, VALUE);

        // Try to add liquidity at exact close time - should fail // NB it doesn't
        vm.warp(closesAt);
        // vm.expectRevert(bytes("!ms"));
        predictionMarket.addLiquidity(marketId, VALUE);
    }
```

Recommended Mitigation:

```
  modifier timeTransitions(uint256 marketId) {
--    if (block.timestamp > markets[marketId].closesAtTimestamp && markets[marketId].state ==
↪  MarketState.open) {
++    if (block.timestamp >= markets[marketId].closesAtTimestamp && markets[marketId].state ==
↪  MarketState.open) {
      _nextState(marketId);
    }
    _;
  }
```

Myriad: Fixed in PR#79, commit `774161e`

Cyrfin: Verified. Check is now >=.

7.1.4 Lack of slippage protection in liquidity functions

Description: The `addLiquidity` and `removeLiquidity` functions do not include any form of slippage protection, such as `minSharesOut` or `minTokensOut` parameters. This means users cannot guard against receiving significantly fewer shares or tokens than expected due to changes in pool state between quoting and execution.

Impact: Without slippage bounds, users are exposed to unfavorable outcomes if the price shifts between the time they calculate expected results off-chain and when the transaction is executed. While this risk is mitigated on chains without public mempools (as currently targeted by the protocol), it becomes relevant on other EVM chains with public transaction visibility, where MEV actors or frontrunners may exploit such gaps.

Proof of Concept: Add the following test to `PredictionMarket.t.sol`:

```
    function testFrontRunAddLiquidity() public {
        uint256 marketId = _createTestMarket();

        address alice = makeAddr("Alice");
        address bob = makeAddr("Bob");
        deal(address(tokenERC20), alice, 1 ether);
        deal(address(tokenERC20), bob, 1 ether);

        // bob sees that alice will add liquidity and buys shares to manipulate the price
        vm.startPrank(bob);
        tokenERC20.approve(address(predictionMarket), type(uint256).max);
        predictionMarket.buy(marketId, 1, 0, 10e16);
        vm.stopPrank();

        // alice adds liquidity
        vm.startPrank(alice);
        tokenERC20.approve(address(predictionMarket), type(uint256).max);
        predictionMarket.addLiquidity(marketId, 1e16);
        vm.stopPrank();

        // bob can now sell for more gaining ~30% of Alice deposit
        vm.prank(bob);
        predictionMarket.sell(marketId, 1, 10.3e16, type(uint256).max);
    }
```

Recommended Mitigation: Consider adding optional slippage parameters to both `addLiquidity` and `removeLiquidity`, e.g.:

```
function addLiquidity(uint256 marketId, uint256 amount, uint256 minSharesOut) external;
function removeLiquidity(uint256 marketId, uint256 shares, uint256 minTokensOut) external;
```

These guardrails would allow users to constrain execution based on expected outcomes and improve safety if the protocol expands to other chains.

Myriad: Fixed in PR#88, commit 37e9e89

Cyfrin: Verified. `addLiquidity` now takes a `minSharesIn` parameter and `removeLiquidity` takes a `minValue` parameter.

7.1.5 Consider adding a `deadline` parameter to user facing calls

Description: The `buy`, `sell`, `addLiquidity`, and `removeLiquidity` functions currently do not accept a `deadline` parameter to constrain when a transaction must be executed. Without a deadline, transactions may be mined at unexpected times, especially if there are delays in submission or relaying, leading to unexpected pricing or market state changes.

Impact: Transactions that execute later than intended may result in unfavorable outcomes for users, particularly in volatile or thin markets. Including a deadline improves predictability and user trust by ensuring the action is executed only within the expected time window.

Recommended Mitigation: Add a `uint256 deadline` parameter to user-facing functions and enforce it using a modifier:

```
modifier onlyBefore(uint256 deadline) {
    require(block.timestamp <= deadline, "!d");
    _;
}
```

This ensures the transaction will revert if not mined before the user-specified deadline.

Myriad: Acknowledged. Even though this extra layer of verification would be helpful, I believe there are already other mechanisms in place to avoid the use cases you mention:

- Slippage is already configurable through the following:
  - `buy` - `minOutcomeSharesToBuy`
  - `sell` - `maxOutcomeSharesToSell`
  - `addLiquidity` - `minSharesIn` (implemented in #88)
  - `removeLiquidity` - `minValueOut` (implemented in #88)

These configs protect users from volatility / unexpected pricing changes.

In order to prevent issues form market state changes, the `timeTransitions` modifier will act and gracefully revert the transaction without the need for a `deadline`

My main concern is mostly adding an extra layer of logic to the main functions, which would add complexity (it would require an extra argument in all functions) - when we do already mechanisms in the function arguments that prevent the mentioned issues.

7.1.6 Consider adding share-based sell function to avoid dust shares

Description: The current interface requires users to specify the desired amount out (tokens) when selling shares, rather than the amount in (shares to sell). This makes it difficult for users or integrators to accurately sell all their shares, especially when prices fluctuate during execution.

In many cases, users must first compute how many tokens they will receive for their full share balance using a view function like `calcSellAmount()`. However, due to price sensitivity and dynamic pool state, this off-chain calculation may become outdated by the time the transaction is submitted.

Impact: Because users cannot sell an exact number of shares, they will likely end up with small leftover dust balances in their accounts. This dust often isn't worth the gas to sell or claim, making it effectively unrecoverable and cluttering user balances over time.

Recommended Mitigation: Consider adding a new function such as:

```
function sellShares(uint256 marketId, uint256 outcome, uint256 shareAmount, uint256 minAmountOut)
↪   external;
```

This would allow users to directly sell a specified number of shares, eliminating off-chain estimation errors and preventing leftover dust.

Myriad: Acknowledged. This makes total sense and we would love to have this option, but sadly it's too complex to implement on a smart contract.

In order to preserve the fixed product market maker formula `(L^n = O * O * ... * On)`, where `L` is the liquidity, `n` is the number of outcomes and `Ox` is the number of shares of outcome `x`, the formula to calculate the amount to be sold in a binary outcome would be the following:

```
// x = token amount user will receive
// shares = shares sold
// S = sell token pool balance
// O = other outcome pool balance

x² - x * (O + S + shares) + shares * O = 0
=> x = 1/2 (-sqrt(O^2 - 2 O (shares + S) + 4 * O * S + (shares + S)^2) + O + shares + S)
```

For a ternary outcome market, it would be a cubic formula:

```
x³ - x²(shares + S + O + O) + x((shares + S)(O + O) + OO) - shares × O × O = 0
```

And so on. The higher the number of outcomes, the more complex the arithmetic expression is. Calculating these is extremely gas-expensive and inaccurate to compute on a smart contract.

The alternative is the method we already have in place - do an offchain estimate of how much are the user's tokens worth, using the Newton-Raphson method, and calling `sell` with the amount returned from the method.

7.2 Informational

7.2.1 Deprecated `testFail` Usage in Tests

Description: Several tests (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) in `PredictionMarket.t.sol` and `PredictionMarketManager.t.sol` use Foundry's `testFail` pattern. This pattern has been deprecated and should be avoided in favor of more explicit revert expectations.

Instead, tests should use `vm.expectRevert()` immediately before the line expected to revert. Ideally, this should include the specific error message to improve clarity and precision: `vm.expectRevert("expected error")`.

Additionally, consider renaming these tests to align with Foundry's best practices, using the `test_Revert[When|If]...` format to clearly convey the revert condition.

Myriad: Fixed in PR#89, commit `dbcfdfa`

Cyfrin: Verified. Tests now catch the expected reverts.

7.2.2 Consider Using `Ownable2StepUpgradeable`

Description: The contracts currently use `OwnableUpgradeable` for access control. To improve safety, consider upgrading to `Ownable2StepUpgradeable`, which enforces an explicit acceptance step for ownership transfers and helps prevent accidental loss of control.

Consider replacing `OwnableUpgradeable` with `Ownable2StepUpgradeable` to enforce a two-step transfer pattern.

In addition, consider overriding the `renounceOwnership()` function to always revert. This prevents accidental or unintended renouncement of ownership, which can leave the contract permanently without an authorized admin.

Myriad: Fixed in PR#83, commit `8efe6e4`

Cyfrin: Verified. `Ownable2StepUpgradeable` is now used.

7.2.3 Consider enforcing a `minAmount` to prevent rounding exploits

Description: To protect against potential precision-based exploits or rounding errors, the protocol should consider enforcing a `minAmount` threshold (e.g., equivalent to $1 in token units) across user-facing actions such as trades, liquidity provision, and claims.

When very small amounts are allowed, they may trigger edge cases in rounding logic, fee calculations, or invariant enforcement, especially in markets using fixed-point math. Even if not currently exploitable, disallowing "dust" amounts provides a safer baseline.

Consider applying a protocol-wide `minAmount` check to user-facing actions such as `buy`, `sell`, `addLiquidity`, and `claim*` functions.

Myriad: Acknowledged. Even though the recommendation makes sense, it's quite complex to implement on a multi-token environment. There's a few points to consider, such as:

- ERC20 Token decimals - The standard is 18, however for stablecoins such as USDC or USDT is 6.
- ERC20 Token Price - Consider DAI and WETH - both have 18 decimals, however 1 DAI = 1$ and 1 WETH > $3700

Given the points above, a protocol-wide `minAmount` wouldn't be appropriate given we might be dealing with different decimals and prices for different markets. In order to implement this, it would have to be enforced on the `PredictionMarketManager` side of things, which in my opinion is an overhead of logic I'd like to avoid at this point.

7.2.4 Disallow single-outcome markets

Description: When creating a market in `PredictionMarketV3_4::_createMarket`, the following check ensures that the number of outcomes is within bounds:

```
require(desc.outcomes > 0 && desc.outcomes <= MAX_OUTCOMES, "!oc");
```

However, allowing `desc.outcomes == 1` results in a prediction market with only one possible outcome, which undermines the purpose of having a market at all.

Consider changing the check to:

```
require(desc.outcomes > 1 && desc.outcomes <= MAX_OUTCOMES, "!oc");
```

to ensure that all created markets include at least two outcomes.

Myriad: Fixed in PR#78, commit `9d0090d`

Cyfrin: Verified. Check for outcomes is not `>= 2`.

7.2.5 Add explicit check to prevent underflow revert in `PredictionMarketV3_4::calcSellAmount`

Description: In `PredictionMarketV3_4::calcSellAmount` following line may revert due to an underflow if the user attempts to sell too many shares:

```
endingOutcomeBalance = (endingOutcomeBalance * outcomeShares).ceilDiv(outcomeShares - amountPlusFees);
```

If `amountPlusFees >= outcomeShares`, the denominator becomes zero or negative, causing a revert. While this is mathematically expected, it may be confusing for users who receive no clear indication of what went wrong.

Consider adding an explicit check such as:

```
require(amountPlusFees < outcomeShares, "a>s");
```

This provides a clearer error message and avoids reverting due to underflow during calculation.

Myriad: Fixed in PR#80, commit `90a2742`

Cyfrin: Verified. A check to verify there's more outcome shares than amount is now in place.

7.2.6 Unused field `MarketResolution.resolved`

Description: The `MarketResolution` struct includes a `resolved` field, but it is not used anywhere in the current contract logic. Keeping unused state variables increases code complexity and may confuse future maintainers or auditors. It may also suggest incomplete or outdated logic.

Consider either removing the `resolved` field if it is unnecessary, or integrate it meaningfully into the resolution logic if it was intended to serve a functional purpose.

Myriad: Fixed in PR#81, commit `6060137`

Cyfrin: Verified. `resolved` removed from the `MarketResolution` struct.

7.2.7 "Weight" and `poolWeight` serves different meanings throughout the code/documentation

Description: The concept weight/variable `poolWeight` is used in multiple places throughout the contract, serving three distinct purposes, which may lead to confusion:

1. In `PredictionMarketV3_4::addLiquidity` and `PredictionMarketV3_4::removeLiquidity`, it represents the minimum or maximum number of outcome shares in the pool.
2. It is also used in `MarketFees.poolWeight` as a fee accumulator
3. In the documentation "weight" is used to represent the product of share outcomes.

Using a single variable name (`poolWeight`) for multiple, semantically different roles can lead to misunderstandings for developers, auditors, and integrators. Also mixing the meaning of the word "weight" in the context of the code obscures the intent behind the logic and may cause misinterpretation of core mechanics like fee behavior, share distribution, or price computation.

Consider separating these concerns by using more specific and descriptive variable names for each use case. For example:

- Use `minOutcomeShares` or `baseShares` for liquidity operations
- Use `feeAccumulator` for fee tracking
- Use `priceWeightProduct` or similar for price-related math

This will improve readability and reduce the cognitive load for maintainers and users of the contract.

Myriad: Fixed in PR#82, commit `d30be85`

Cyfrin: Verified. `poolWeight` in `MarketFees` renamed to `feeAccumulator` and variable `poolWeight` renamed to `baseShares`.

7.2.8 Unnecessary transfer in `PredictionMarketV3_4::mintAndCreateMarket`

Description: `PredictionMarketV3_4::mintAndCreateMarket` does a mint of `FantasyERC20` to the user then a transfer to the contract:

```
function mintAndCreateMarket(CreateMarketDescription calldata desc) external nonReentrant returns
↪   (uint256 marketId) {
  // mint the amount of tokens to the user
  IFantasyERC20(address(desc.token)).mint(msg.sender, desc.value);

  marketId = _createMarket(desc);
  // transferring funds
  desc.token.safeTransferFrom(msg.sender, address(this), desc.value);

  return marketId;
}
```

The issue is that the second transfer is unnecessary as first `desc.value` is minted to the user, then `desc.value` is transferred to the contract. Consider simplifying this to:

```
function mintAndCreateMarket(CreateMarketDescription calldata desc) external nonReentrant returns
↪   (uint256 marketId) {
  // mint the amount of tokens to the user
  IFantasyERC20(address(desc.token)).mint(address(this), desc.value);

  return _createMarket(desc);
}
```

This saves a transfer and doesn't require the user to approve the contract to spend `FantasyERC20` tokens.

Myriad: Fixed in PR#86, commit `ba955d8`

Cyfrin: Verified. Mint is now done directly to the contract and transfer removed.

7.2.9 Missing event for important state changes

Description: The following calls doesn't emit any events:

- `LandFactory::updateLockAmount`
- `PredictionMarketV3_4::withdraw`

Events enable off-chain tracking, auditing and transparency. Consider emitting events from the calls above.

Myriad: Fixed in PR#87, commits `b40b740` and `52bdf82`

Cyfrin: Verified. Both calls now emit events.