Pashov Audit Group

Memecoin Prediction Markets Security Review

August 7th 2025 - August 8th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

• **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
• **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
• **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

• **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
• **Medium** - only a conditionally incentivized attack vector, but still relatively likely
• **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Memecoin Prediction Markets

Memecoin Prediction Markets is a smart contract that lets users predict whether the price of a crypto token will rise or fall in a given round, automatically tracking stakes, outcomes, and rewards. It manages predictions and distributes winnings.

5. Executive Summary

A time-boxed security review of the **ilijasdev/mcp-market** repository was done by Pashov Audit Group, during which Pashov Audit Group engaged to review **Memecoin Prediction Markets**. A total of **12** issues were uncovered.

Protocol Summary	
Project Name	Memecoin Prediction Markets
Protocol Type	Prediction outcome manager
Timeline	August 7th 2025 - August 8th 2025

Review commit hash:

- bdf0e02329fc240186f93ad6ffc5ea6c0474f88f (ilijasdev/mcp-market)

Fixes review commit hash:

- 562d0079a053138bfe1a6c6b82ad4a7d2dcfe103 (ilijasdev/mcp-market)

Scope

`memecoin-predictions-01.sol`

6. Findings

Findings count

Severity	Amount
High	1
Medium	3
Low	8
Total findings	**12**

Summary of findings

ID	Title	Severity	Status
[H-01]	`claimReward()` can DOS by iterating predictions blocking funds	High	**Resolved**
[M-01]	Funds lock if all users choose same direction and price is incorrect	Medium	**Resolved**
[M-02]	Late participation advantage in `MemePredictionMarket`	Medium	**Resolved**
[M-03]	Single-prediction design raises gas costs for high-volume users	Medium	**Resolved**
[L-01]	Later prediction rounds may not start at expected timestamp	Low	**Resolved**
[L-02]	Some assets may be locked in the contract if treasury is address(0)	Low	**Resolved**
[L-03]	Blacklisted users' tokens are stuck after a win	Low	**Resolved**
[L-04]	Bearish users have an advantage over bullish ones	Low	Acknowledged
[L-05]	Missing events in `endMultipleRounds`	Low	**Resolved**
[L-06]	Stale price data may cause incorrect round outcomes	Low	**Resolved**
[L-07]	New rounds can start while current round is active, trapping funds	Low	**Resolved**
[L-08]	`endMultipleRounds()` lacks round end timestamp check	Low	Acknowledged

High findings

[H-01] `claimReward()` can DOS by iterating predictions blocking funds

Severity

Impact: High

Likelihood: Medium

Description

Users can create an unlimited number of predictions for a specific market and round using `predict()`.

```
function predict(uint256 marketId, bool isBullish) external roundActive(marketId) {
        uint256 roundId = currentRoundId[marketId];
        Round memory round = marketRounds[marketId][roundId];

        // Predictions close 1 hour before round ends to prevent last-minute manipulation
        require(block.timestamp < round.startTime + 23 hours, "Prediction closed (last hour)");

        // Transfer 5 USDC fee from user to contract
        require(IERC20(markets[marketId].token).transferFrom(
            msg.sender,
            address(this),
            PREDICTION_FEE
        ), "Payment failed");

        // Store the prediction
        predictions[marketId][roundId].push(Prediction({
            user: msg.sender,
            isBullish: isBullish,
            amount: PREDICTION_FEE
        }));
        emit PredictionMade(marketId, roundId, msg.sender, isBullish, PREDICTION_FEE);
    }
```

The `predictions` array can grow indefinitely because it is unbounded. When `claimReward()` is called, the entire `predictions` array is checked.

```
function claimReward(uint256 marketId, uint256 roundId) external {

        Round memory round = marketRounds[marketId][roundId];
        require(round.ended, "Round not ended");
        require(!hasClaimed[marketId][roundId][msg.sender], "Already claimed");

        Prediction[] memory preds = predictions[marketId][roundId];

        // Determine winning direction (bullish if end price > start price)
        bool isBullWinning = round.endPrice > round.startPrice;

        // Calculate reward pool (total - 5% fee)
        uint256 totalPool = preds.length * PREDICTION_FEE;
        uint256 fee = (totalPool * 5) / 100;
        uint256 rewardPool = totalPool - fee;

        // Calculate user's share of winning predictions
        uint256 totalWinningAmount = 0;
        uint256 userWinningAmount = 0;

        for (uint256 i = 0; i < preds.length; i++) {
            if (preds[i].isBullish == isBullWinning) {
                totalWinningAmount += preds[i].amount;
                if (preds[i].user == msg.sender) {
                    userWinningAmount += preds[i].amount;
                }
            }
        }

       ///code...
    }
```

This could cause the transaction to exceed the block’s maximum gas limit and revert, resulting in a denial-of-service (DoS) that permanently locks funds in the contract.

This scenario is likely because the amount is fixed at 5 USDC in `PREDICTION_FEE`. For example, if a user wants to predict 10,000 USDC, 2,000 entries would be added to the `predictions` array.

Recommendations

To solve the issue, set a maximum limit on the number of predictions allowed per round.

Medium findings

[M-01] Funds lock if all users choose same direction and price is incorrect

Severity

Impact: High

Likelihood: Low

Description

The only way to withdraw tokens from the contract (excluding the fees sent to the treasury) is through `claimReward()`.

```
function claimReward(uint256 marketId, uint256 roundId) external {
        Round memory round = marketRounds[marketId][roundId];
        require(round.ended, "Round not ended");
        require(!hasClaimed[marketId][roundId][msg.sender], "Already claimed");

        Prediction[] memory preds = predictions[marketId][roundId];

        // Determine winning direction (bullish if end price > start price)
        bool isBullWinning = round.endPrice > round.startPrice;

        // Calculate reward pool (total - 5% fee)
        uint256 totalPool = preds.length * PREDICTION_FEE;
        uint256 fee = (totalPool * 5) / 100;
        uint256 rewardPool = totalPool - fee;

        // Calculate user's share of winning predictions
        uint256 totalWinningAmount = 0;
        uint256 userWinningAmount = 0;

        for (uint256 i = 0; i < preds.length; i++) {
            if (preds[i].isBullish == isBullWinning) {
                totalWinningAmount += preds[i].amount;
                if (preds[i].user == msg.sender) {
                    userWinningAmount += preds[i].amount;
                }
            }
        }

        require(userWinningAmount > 0, "Not in winning group");

        // Calculate user's reward based on their share
        uint256 reward = (rewardPool * userWinningAmount) / totalWinningAmount;

        // Mark as claimed and transfer reward
        hasClaimed[marketId][roundId][msg.sender] = true;
        IERC20(markets[marketId].token).transfer(msg.sender, reward);
        emit RewardClaimed(marketId, roundId, msg.sender, reward);
    }
```

As seen in the code, calculating a user's reward involves iterating through all predictions to determine `totalWinningAmount` and `userWinningAmount`. If all users predict the same direction, `totalWinningAmount` and `userWinningAmount` become zero for all the users. This results in no one being able to claim rewards, effectively locking all funds in the contract permanently.

If even a single user predicts the opposite direction, they receive the entire `rewardPool`. However, if no one does, there's no valid winning side, and since there's no alternative mechanism to withdraw the tokens, they remain permanently stuck in the contract with no way to recover them.

Recommendations

One solution could be to send all the tokens from that round to the treasury if no one correctly predicts the price direction.

[M-02] Late participation advantage in `MemePredictionMarket`

Severity

Impact: Medium

Likelihood: Medium

Description

The current design of MemePredictionMarket allows users to make predictions for most of the round's duration (e.g., a 24-hour round with prediction open for 23 hours), which unintentionally benefits late participants. These users can observe more price action and make better-informed decisions than early participants, leading to an unfair advantage.

Example scenario:

- At the beginning of the round, the token price is 10.
- A user joins early and makes a prediction.
- Near the deadline (e.g., hour 23), the price rises to 15.
- A new user sees this trend and makes the same bullish prediction, with a much higher confidence of winning.

This system disproportionately rewards those who join late, potentially disincentivizing early participation and harming the fairness of the prediction market.

Recommendations

Implement time-based weighting: Earlier predictions could receive higher weight or rewards to compensate for increased uncertainty.

incorporate price delta-based weighting: If a user joins after a significant price movement (e.g., from 10 to 20), their prediction reward could be adjusted based on the reduced risk taken.

[M-03] Single-prediction design raises gas costs for high-volume users

Severity

Impact: Medium

Likelihood: High

Description

The current predict function in the MemePredictionMarket contract only supports a fixed single prediction per transaction. Each user is charged exactly 5 USDC (PREDICTION_FEE) for every prediction they make, with no way to scale the amount in a single call.

This creates a bottleneck for users who want to place large predictions (e.g., $1,000 or $10,000). These users must call the predict function multiple times, each transferring 5 USDC, resulting in repetitive token transfer operations and expensive gas fees, especially on high-congestion chains like Ethereum Mainnet. When gas prices spike, each call may cost over $5 in gas, making the user experience inefficient and economically unreasonable.

Recommendations

Allow users to place multiple predictions in a single transaction

```
function predict(uint256 marketId, bool isBullish, uint256 amount) external
roundActive(marketId) {

// Total prediction cost
uint256 totalFee = PREDICTION_FEE * amount;

// Transfer total USDC fee from user to contract
require(
    IERC20(markets[marketId].token).transferFrom(
        msg.sender,
        address(this),
        totalFee
    ),
    "Payment failed"
);
```

```
// Store prediction
predictions[marketId][roundId].push(Prediction({
    user: msg.sender,
    isBullish: isBullish,
    amount: totalFee
}));
```

Low findings

[L-01] Later prediction rounds may not start at expected timestamp

According to the discussion with sponsors, we expect to start a new round at the expected timestamp. For example, we expect to start a new round in each UTC 00:00.

The problem here is that when the owner wants to end this round at start timestamp + 24 hours, it will be quite difficult to end the round exactly at start timestamp + 24 hours. It's quite possible that we may delay one or two blocks. Then we can start one new round after the start timestamp + 24 hours + several blocks. In each round prediction, we will add one delay here. When we finish hundreds of rounds, we will find out that the actual start time for one new round is far away from our expected UTC 00:00. This will have a bad influence on predictors.

```
function endRound(uint256 marketId) external onlyOwner {
    uint256 roundId = currentRoundId[marketId];
    // storage.
    Round storage round = marketRounds[marketId][roundId];
    require(!round.ended, "Round already ended");
    // If we reach the end of this round, the owner can end one round.
    require(block.timestamp >= round.startTime + 24 hours, "Round not over yet");
}
```

Recommendation: Remove this requirement.

[L-02] Some assets may be locked in the contract if treasury is address(0)

When we end one round, we will calculate the fee. And based on current logic, we will transfer the treasury fee to the treasury address if the treasury address is not address(0).

The problem here is that if the treasury address is address(0), we will not transfer the treasury fee, and when users claim rewards, this part of fees will still be deducted from the rewards. So this part of the assets will be locked.

```
function endRound(uint256 marketId) external onlyOwner {
    ...
    // Calculate and transfer 5% fee to treasury
    // total predict asset for this round.
    uint256 totalPool = predictions[marketId][roundId].length * PREDICTION_FEE;
    // treasury fee.
    uint256 fee = (totalPool * 5) / 100;

    if (fee > 0 && treasury != address(0)) {
        // the token will always be USDC.
        IERC20(markets[marketId].token).transfer(treasury, fee);
        emit FeeTransferred(marketId, roundId, treasury, fee);
    }
    emit RoundEnded(marketId, roundId, price, totalPool, fee);
}
```

Recommendation: Here we have one assumption that the `treasury` should not be address(0). Suggest adding one security check to make sure that `treasury` is not address(0).

[L-03] Blacklisted users' tokens are stuck after a win

To make predictions, `predict()` is used with USDC tokens, which implement blacklist logic. If a user is blacklisted after making a prediction and wins, their tokens will be permanently stuck in the contract because `claimReward()` sends rewards only to the original user, who can no longer withdraw due to the blacklist.

Recommendation: Implement an owner-only function to withdraw USDC in such cases to prevent funds from being locked forever.

[L-04] Bearish users have an advantage over bullish ones

To determine whether the prediction is bullish or bearish in `claimReward()`, the calculation is as follows:

```
bool isBullWinning = round.endPrice > round.startPrice;
```

If `round.endPrice == round.startPrice`, the bearish predictions win by default. This gives the bearish position a slight advantage over the bullish.

Recommendation: If `round.endPrice == round.startPrice`, refund the entire deposited amount to the users.

[L-05] Missing events in `endMultipleRounds`

The endRound function emits RoundEnded and FeeTransferred events, which are crucial for off-chain services track contract activity. The batch function endMultipleRounds performs the same logic but does not emit these events. This creates an inconsistency and makes it difficult for dApps and monitoring tools to track rounds that are closed via the batch function.

Add the corresponding emit statements inside the loop in endMultipleRounds.

[L-06] Stale price data may cause incorrect round outcomes

In the endRound function, the contract retrieves the latest value from the DIA price feed, but does not check whether the price is fresh or from before the round's end time (startTime + 24 hours). As a result, it is possible for the oracle to return a stale price. This creates a fairness issue, where the final round result (and thus reward distribution) might not accurately reflect the market conditions at the end of the round, potentially disadvantaging users who made correct predictions.

Add a check that ensures the returned price timestamp is after or at the round's expected end time.

[L-07] New rounds can start while current round is active, trapping funds

`startNewRound()` is used to initiate a new round for a specific market.

```
function startNewRound(uint256 marketId) external onlyOwner {
        require(marketId < marketCount, "Invalid market");

        // Get current price from DIA price feed
        (uint128 price,) = getPrice(markets[marketId].tokenKey, markets[marketId].priceFeed);

        // Increment round counter and create new round
        currentRoundId[marketId]++;
        marketRounds[marketId][currentRoundId[marketId]] = Round({
            startTime: block.timestamp,
            startPrice: price,
            endPrice: 0,
            ended: false
        });
        emit RoundStarted(marketId, currentRoundId[marketId], block.timestamp, price);
    }
```

As you can see, it does not check if a round is currently ongoing and increases `currentRoundId`. The issue is that `endRound()` relies on the stored `currentRoundId`, which may now point to an unfinished round.

```
function endRound(uint256 marketId) external onlyOwner {
@>      uint256 roundId = currentRoundId[marketId];
        Round storage round = marketRounds[marketId][roundId];
        require(!round.ended, "Round already ended");
        require(block.timestamp >= round.startTime + 24 hours, "Round not over yet");

        // Get final price from DIA
        (uint128 price,) = getPrice(markets[marketId].tokenKey, markets[marketId].priceFeed); /

        // Set final price and mark round as ended
        round.endPrice = price;
        round.ended = true;

        // Calculate and transfer 5% fee to treasury
        uint256 totalPool = predictions[marketId][roundId].length * PREDICTION_FEE;

        uint256 fee = (totalPool * 5) / 100;

        if (fee > 0 && treasury != address(0)) {
            IERC20(markets[marketId].token).transfer(treasury, fee);
            emit FeeTransferred(marketId, roundId, treasury, fee);
        }
        emit RoundEnded(marketId, roundId, price, totalPool, fee);
    }
```

This causes a problem where, if `startNewRound()` is called while a round is still active and some users have already deposited tokens using `predict()`, their funds become permanently stuck in the contract.

This is because the only way to withdraw these funds is through `claimReward()`, which depends on `endRound()` being called to set `round.ended = true` for that round.

```
function claimReward(uint256 marketId, uint256 roundId) external {
        Round memory round = marketRounds[marketId][roundId];
@>      require(round.ended, "Round not ended");
        require(!hasClaimed[marketId][roundId][msg.sender], "Already claimed");

///code...
}
```

In this case, it will be impossible to set `round.ended = true` because the new round has already started and the previous round cannot be ended, causing all funds from that round to be permanently locked in the contract.

Recommendations

To solve the problem, check if a round is currently ongoing for the market and only allow starting a new round once the current round's `ended` status is set to true. The same check should be applied to `startMultipleRounds()`.

```
function startNewRound(uint256 marketId) external onlyOwner {
        require(marketId < marketCount, "Invalid market");
+        Round memory currentRound = marketRounds[marketId][currentRoundId[marketId]];

+        if (currentRound.startTime != 0) {
+            require(currentRound.ended, "Previous round not finshed");
+        }

        // Get current price from DIA price feed
        (uint128 price,) = getPrice(markets[marketId].tokenKey, markets[marketId].priceFeed);

        // Increment round counter and create new round
        currentRoundId[marketId]++;
        marketRounds[marketId][currentRoundId[marketId]] = Round({
            startTime: block.timestamp,
            startPrice: price,
            endPrice: 0,
            ended: false
        });
        emit RoundStarted(marketId, currentRoundId[marketId], block.timestamp, price);
    }
```

[L-08] `endMultipleRounds()` lacks round end timestamp check

`endMultipleRounds()` is used to end multiple rounds across different markets in a single transaction.

```
function endMultipleRounds(uint256[] calldata marketIds) external onlyOwner {
        for (uint256 i = 0; i < marketIds.length; i++) {
            uint256 marketId = marketIds[i];
            uint256 roundId = currentRoundId[marketId];
            Round storage round = marketRounds[marketId][roundId];

            if (!round.ended) {
                (uint128 price,) = getPrice(markets[marketId].tokenKey,
markets[marketId].priceFeed);
                round.endPrice = price;
                round.ended = true;

                uint256 totalPool = predictions[marketId][roundId].length * PREDICTION_FEE;
                uint256 fee = (totalPool * 5) / 100;

                if (fee > 0 && treasury != address(0)) {
                    IERC20(markets[marketId].token).transfer(treasury, fee);
                }
            }
        }
    }
```

As you can see, it does not verify whether the end timestamp has been reached before ending the round, allowing it to be ended at any time and bypassing the 24-hour window.

Recommendations

Checks if the round's end timestamp has been reached; if not, it reverts.

```
function endMultipleRounds(uint256[] calldata marketIds) external onlyOwner {
        for (uint256 i = 0; i < marketIds.length; i++) {
            uint256 marketId = marketIds[i];
            uint256 roundId = currentRoundId[marketId];
            Round storage round = marketRounds[marketId][roundId];

+           require(block.timestamp >= round.startTime + 24 hours, "Round not over yet");

            if (!round.ended) {
                (uint128 price,) = getPrice(markets[marketId].tokenKey,
markets[marketId].priceFeed);
                round.endPrice = price;
                round.ended = true;
```