The Accelerated Curve

Nirlin Solo Audit

PNP Prediction Market Audit Report

June 21, 2025

This audit report presents a focused security assessment of the PNP Prediction Market, highlighting potential vulnerabilities and areas for improvement.

Scope and Summary of Findings

Scope of Audit

Contract Files Audited
PNPFactory.sol
libraries/PythagoreanBondingCurve.sol

Summary of Findings

High Severity

Title	Severity
Erroneous Calculation Causes Inconsistent Reserve Initialization	High
Fee Is Never Accrued and Cannot Be Claimed	High

Medium Severity

Title	Severity
Condition ID Can Be Front-Run to Deny Legitimate Market Creation	Med

Low Severity

Title	Severity
NO Tokens Are Not Burned During Redemption	Low

PNPFactory: Initial Liquidity Causes Inconsistent Reserve Initialization

Summary

During market creation, the PNPFactory contract mints equal amounts of YES and NO tokens based on the provided initial liquidity. However, the reserve is incorrectly initialized to the raw liquidity amount, which violates the bonding curve's assumptions and causes mispricing in all future operations.

Description

The bonding curve expects that:

```
reserves = sqrt(yes^2 + no^2)
```

But the contract sets:

```
marketReserve[conditionId] = scaledLiquidity;
```

If 10 USDC is provided, the contract mints 10 YES and 10 NO tokens, but the reserve is recorded as 10 instead of 14.14. This misalignment distorts all mint, burn, and redeem logic from the start.

Proof of Concept

Relevant code snippet from ‘createPredictionMarket‘:

```
marketReserve[conditionId] = scaledLiquidity; // incorrect reserve
_mint(msg.sender, yesTokenId, scaledLiquidity, "");
_mint(msg.sender, noTokenId, scaledLiquidity, "");
```

Expected Results with Vulnerable Code

```
Initial Liquidity: 10 USDC
YES Tokens: 10
NO Tokens: 10
Reserve: 10   // wrong
```

Expected Results with Fix Applied

```
Initial Liquidity: 10 USDC
YES Tokens: 10
NO Tokens: 10
Reserve: 14.14   // correct
```

Recommendation

Refactor ‘createPredictionMarket‘ to compute the reserve using the bonding curve formula:

PNPFactory: Take-Fee Logic Does Not Remove Fees

Summary

The ‘mintDecisionTokens‘ function computes a post-fee amount for pricing but then adds the full pre-fee collateral into the reserve. As a result, the intended fee is never separated, distorting the bonding curve state and giving winners more collateral than intended.

Description

In ‘mintDecisionTokens‘, the contract does:

```
uint256 amountAfterFee = (collateralAmount * (10000 - TAKE_FEE)) / 10000;
uint256 scaledAmount   = scaleTo18Decimals(amountAfterFee, collateralDecimals);
// ... tokensToMint is calculated from scaledAmount ...
IERC20(collateralToken[conditionId]).transferFrom(msg.sender, address(this),
    collateralAmount);
marketReserve[conditionId] = scaledReserve + scaledFullAmount;
```

Here, ‘amountAfterFee‘ (e.g., 99% of collateral) is used to price new tokens, but the code then adds ‘scaledFullAmount‘ (100% of collateral) to ‘marketReserve‘. The 1% “fee” never leaves the reserve—no address or separate bucket collects it. This misalignment breaks all subsequent minting, burning, and redemption calculations, and winners end up receiving the “fee” in addition to their rightful share.

Proof of Concept

```
When ‘TAKE_FEE = 100‘ (1\%), and a user supplies 100 USDC:
```

- ‘amountAfterFee = 99 USDC‘ → pricing uses 99.
- But ‘transferFrom‘ moves 100 USDC, and ‘marketReserve += 100‘.
- The extra 1 USDC silently inflates the curve’s reserve.

As a result, the next minter pays 100 USDC but receives tokens priced as if only 99 USDC entered, and that extra 1 USDC remains in ‘marketReserve‘ without ever going to the fee recipient.

Recommendation

Adjust ‘mintDecisionTokens‘ so that only the post-fee portion is added to ‘marketReserve‘, and track or distribute the fee separately. For example:

```
// After transferFrom(msg.sender, address(this), collateralAmount):
uint256 scaledFull   = scaleTo18Decimals(collateralAmount, collateralDecimals);
uint256 scaledPost   = scaleTo18Decimals(amountAfterFee, collateralDecimals);
uint256 scaledFee    = scaledFull - scaledPost;

// Only add scaledPost to the reserve:
marketReserve[conditionId] = scaledReserve + scaledPost;

// Accumulate or send the fee elsewhere:
accumulatedFees[conditionId] += scaledFee;
```

Ensure that ‘accumulatedFees‘ (or a designated fee-recipient) is paid out appropriately when the market settles. This preserves the bonding-curve invariants and ensures the 1% fee is actually collected.

Collected Fees Are Not Withdrawable

Summary

The contract charges a fee on each collateral deposit via `TAKE_FEE`, but provides no function or mechanism to withdraw or claim the accumulated fees. This results in fees being locked permanently in the reserve.

Description

Whenever a user mints decision tokens, a fee is deducted:

```
uint256 amountAfterFee = (collateralAmount * (10000 - TAKE_FEE)) / 10000;
```

However, the deducted portion is not tracked or stored separately. There is no function like `claimFees()` or any dedicated mapping for fee balances. As a result, the protocol has no way to extract or utilize the collected fees.

Impact

High — Fee logic exists and users are charged, but the value is unrecoverable. This leads to protocol revenue being effectively burned and undermines intended fee-based incentives.

Recommendation

- Track collected fees in a dedicated variable or mapping.
- Introduce a `claimFees()` function callable by the owner or designated address.
- Emit events for better fee tracking and accounting.

NO Tokens Are Not Burned During Redemption

Summary

The current redemption flow only burns YES tokens when a user redeems for collateral. The NO tokens remain in the user’s wallet, resulting in skewed accounting and potential confusion for users or integrators.

Description

When a user redeems their position, the contract burns only the YES tokens:

```
// Burn YES tokens
_burn(msg.sender, yesTokenId, amount);
```

The corresponding NO tokens remain untouched in the user's wallet. Since both YES and NO tokens were minted together and the bonding curve assumes their supply reflects market position, leaving NO tokens unburned may distort post-resolution analytics or integrations.

Impact

Low — This does not cause fund loss or security failure but creates a minor logic inconsistency and potential UX confusion. Over time, it may lead to ghost NO token balances that have no utility.

Recommendation

- Burn both YES and NO tokens during redemption to reflect full position closure.
- Alternatively, document clearly that users must manually dispose of NO tokens.

PNPFactory: Condition ID Can Be Front-Run to Deny Legitimate Market Creation

Summary

Because the contract derives `conditionId` solely from the question text and `_endTime`, anyone observing the mempool can front-run `createPredictionMarket`. An attacker can submit the same parameters with minimal liquidity, set `isMarketCreated[conditionId] = true`, and cause the original transaction to revert, effectively performing a denial-of-service (DoS) against the rightful market creator.

Description

- **Derivation.** `conditionId = keccak256(abi.encodePacked(_question, _endTime))`.
- **Permissionless call.** Anyone can call `createPredictionMarket`; there is no access control or anti-front-running guard.
- **DoS vector.** The first transaction that lands with the chosen `conditionId` flips `isMarketCreated` to `true`. Subsequent identical calls revert with `Invalid Market`.

Proof of Concept

```
// Alice's intended tx (in mempool)
createPredictionMarket(
    1_000 * 1e6,                  // 1 000 USDC
    USDC,
    "Will BTC exceed $100k by 1 Jan 2026?",
    1767225600                   // 1 Jan 2026
);

// Attacker spots tx, copies params, but uses 1 USDC and higher gas
createPredictionMarket(
    1 * 1e6,                      // 1 USDC
    USDC,
    "Will BTC exceed $100k by 1 Jan 2026?",
    1767225600
);
```

```
16
17 // Attacker's tx added first      isMarketCreated[conditionId] = true
18 // Alice's tx now reverts
```

Recommendation

- Include the `msg.sender` (or another unique salt) in the `conditionId` hash, e.g. `keccak256(abi.encodePacke`
  `_endTime, msg.sender))`, so only the initiator can reuse that ID.