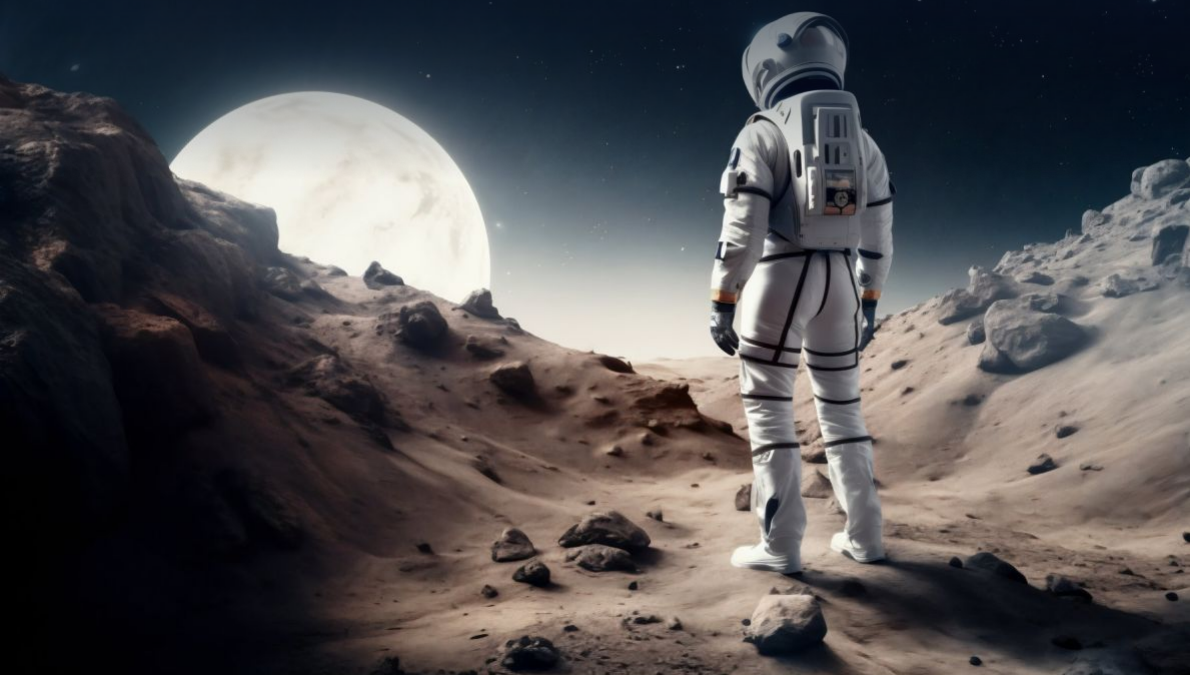
AstraSec

Pandora Protocol Security Audit Report

December 27, 2025

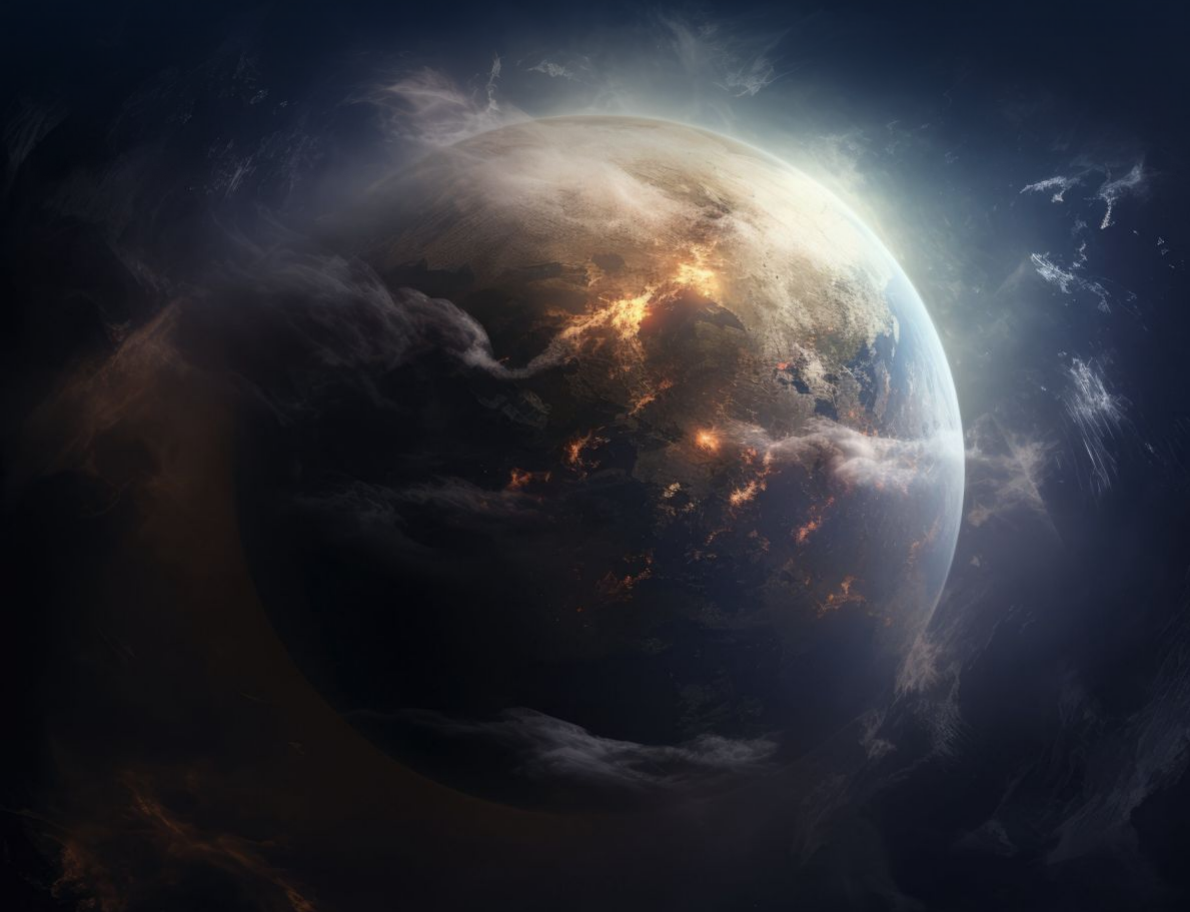


Contents

1 Introduction

1.1 About Pandora Protocol

Pandora Protocol is a decentralized prediction market protocol enabling users to create and trade binary outcome markets on Sonic Network. The system supports both constant product AMM and pari-mutuel market mechanisms, with automated pricing, liquidity provision, and oracle-based resolution for efficient market operations.



1.2 Source Code

The following source code was reviewed during the audit:

- https://github.com/RealWagmi/prediction-oracle-contracts.git
- CommitID: cda3627

And this is the final version representing all fixes implemented for the issues identified in the audit:

- https://github.com/RealWagmi/prediction-oracle-contracts.git
- CommitID: 1bb7e2b

Please note this audit focuses on the security of the contracts themselves. The correctness and reasoning of the economic model itself are not within the scope of this audit.

1.3 Revision History

Version	Date	Description
v1.0	December 27, 2025	Initial Audit

2 Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the Pandora protocol. Throughout this audit, we identified a total of 7 issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	–	–	–	–
High	1	–	–	1
Medium	3	1	–	2
Low	3	–	–	3
Informational	–	–	–	–
Undetermined	–	–	–	–

3 Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

Severity	Acknowledged
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and re-mediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

3.3 Vulnerability Details

3.3.1 [H-1] Potential Sandwich/MEV Attack in PredictionAMM::addLiquidity()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
PredictionAMM.sol	Time and State	High	High	Addressed

The addLiquidity() function lacks slippage protection for subsequent liquidity providers, making it vulnerable to MEV/sandwich attacks. When poolShareSupply is larger than 0, the function calculates proportional amounts (amounts.yesToAdd and amounts.noToAdd) based on current reserves without allowing users to specify minimum acceptable ratios. An attacker can front-run the user's transaction by executing trades that deliberately imbalance the pool, forcing the user to add liquidity at an unfavorable ratio. After the user's transaction, the attacker can back-run by executing reverse trades to restore the original balance, profiting from the price movement while the user's LP position is locked at the manipulated ratio.

prediction-oracle-contracts - PredictionAMM.sol

```
246 function addLiquidity(
247     uint256 collateralAmt,
248     uint256[2] calldata distributionHint,
249     uint256 deadline
250 ) external nonReentrant checkDeadline(deadline) returns (uint256 mintAmount)
251 {
252     ...
253     if (poolShareSupply == 0) {
254         ...
255     } else {
256         ...
257         // Calculate proportional amounts
258         uint256 poolWeight = Math.max(reserves.rYes, reserves.rNo);
259         // Normalize to 18 decimals for LP tokens
260         // poolShareSupply is in 18 decimals, poolWeight is in collateral decimals
261         mintAmount = Math.mulDiv( collateralAmt * lpPrecision, poolShareSupply, poolWeight * lpPrecision);
262
263         // Calculate what to add to maintain proportion
264         amounts.yesToAdd = Math.mulDiv(collateralAmt, reserves.rYes, poolWeight);
265         amounts.noToAdd = Math.mulDiv(collateralAmt, reserves.rNo, poolWeight);
266
267         // Calculate excess to return
268         amounts.yesToReturn = collateralAmt - amounts.yesToAdd;
269         amounts.noToReturn = collateralAmt - amounts.noToAdd;
270     }
271     ...
272 }
```

Remediation Add minYesToAdd and minNoToAdd parameters to provide slippage protection

3.3.2 [M-1] Missing Price Volatility Check for Hour-Boundary Trades

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
PredictionAMM.sol	Business Logic	High	High	Addressed

The _checkAndUpdatePrice() function lacks price volatility validation when crossing hour boundaries, creating a window for price manipulation attacks. When storedHour != currentHour (indicating a new hourly window), the function unconditionally resets the price window by setting anchorPrice, minPrice, and maxPrice to the current newPrice without validating whether this price exceeds the maximum allowed swing relative to the previous hour's price range. An attacker can exploit this by timing large trades exactly at the hour boundary, causing significant price movements that would normally be rejected by the volatility check, effectively bypassing the hourly price swing protection mechanism.

prediction-oracle-contracts-main - PredictionAMM.sol

```
1845 function _checkAndUpdatePrice(uint64 newPrice) internal {
1846     uint256 maxAllowedSwing = _maxPriceSwingScaled;
1847     if (maxAllowedSwing == 0) return; // Limit disabled
1848
1849     // Load and unpack
1850     uint256 data = _priceWindow;
1851     uint32 storedHour = uint32(data & HOUR_MASK);
1852     uint64 anchorPrice = uint64((data >> ANCHOR_OFFSET) & PRICE64_MASK);
1853     uint64 minPrice = uint64((data >> MIN_OFFSET) & PRICE64_MASK);
1854     uint64 maxPrice = uint64((data >> MAX_OFFSET) & PRICE64_MASK);
1855
1856     uint32 currentHour = uint32(_blockTimestamp() / WINDOW_DURATION);
1857
1858     if (storedHour != currentHour) {
1859         // New hour: reset window
1860         storedHour = currentHour;
1861         anchorPrice = newPrice;
1862         minPrice = newPrice;
1863         maxPrice = newPrice;
1864     } else {
1865         ...
1866     }
1867     ...
1868         (uint256(minPrice) << MIN_OFFSET) | (uint256(maxPrice) << MAX_OFFSET);
1869 }
```

Remediation Add a volatility check when resetting the window at hour boundaries by comparing the new price against the previous hour's price range.

3.3.3 [M-2] Revisited Logic of PredictionAMM::_getEffectiveBaseline()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
PredictionAMM.sol	Business Logic	High	High	Addressed

The _getEffectiveBaseline() function ignores the yesToNo parameter when crossing hour boundaries, causing incorrect baseline price calculation. When storedHour != currentHour, the function always returns YES price regardless of swap direction. For NO→YES swaps (yesToNo = false), the baseline should be the NO price (ONE - YES price), not the YES price, leading to incorrect maximum allowed swap amount calculations at hour boundaries.

prediction-oracle-contracts-main - PredictionAMM.sol

```
1902 function _getEffectiveBaseline(
1903     bool yesToNo,
1904     Reserves memory reserves
1905 ) internal view returns (uint64 effectiveBaseline) {
1906     uint256 data = _priceWindow;
1907     uint32 storedHour = uint32(data & HOUR_MASK);
1908     uint32 currentHour = uint32(_blockTimestamp() / WINDOW_DURATION);
1909
1910     if (storedHour != currentHour) {
1911         // New hour - use current price as baseline
1912         return _priceYesFromReserves(reserves);
1913     }
1914
1915     ...
1916 }
```

Remediation Update _getEffectiveBaseline() to consider the yesToNo parameter when crossing hour boundaries: return the YES price for YES→NO swaps and ONE - YES price for NO→YES swaps.

3.3.4 [M-3] Potential Risks Associated with Centralization

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Security	High	Low	Acknowledged

In the Pandora protocol, the existence of a privileged owner account introduces centralization risks, as it holds significant control and authority over critical operations governing the protocol. In the following, we show the representative function potentially affected by the privileges associated with the privileged account.

prediction-oracle-contracts-main - MarketFactory.sol

```
166 function setOutcomeTokenImplementation(
167     address _implementation
168 ) external override onlyOwner {
169     address oldImplementation = outcomeTokenImplementation;
170     outcomeTokenImplementation = _implementation;
171 
172     emit ImplementationUpdated(
173         "OutcomeToken",
174         oldImplementation,
175         _implementation
176     );
177 }
```

Remediation To mitigate the identified issue, it is recommended to introduce multi-sig mechanism to undertake the role of the privileged account. Moreover, it is advisable to implement timelocks to govern all modifications to the privileged operations.

Response By Team This issue has been confirmed by the team.

3.3.5 [L-1] Improved Logic of PredictionAMM::initializer()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Business Logic	Medium	Low	Addressed

In the initializer() modifier of PredictionAMMM, OutcomeToken, PredictionPairMutuel and PredictionPoll, the contract checks if (_initialized) and reverts if already set, but only marks _initialized = true after the external calls and state-changing logic inside initialize().
If any part of the initialize() implementation makes an external call to an attacker-controlled address that supports reentrancy hooks, the attacker can re-enter initialize() while _initialized is still false. This allows multiple initializations in the same transaction.

The vulnerability exists because the effect (setting _initialized = true) is performed after the potentially reentrant interaction, violating the Checks-Effects-Interactions pattern.

prediction-oracle-contracts-main - PredictionAMM.sol

```
116 /// @notice Restrict to one-time initialization (for clone pattern)
117 modifier initializer() {
118     if (_initialized) revert AlreadyInitialized();
119     _;
120     _initialized = true;
121 }
```

Remediation Move _initialized = true; to the very beginning of the initializer modifier, right after the if (_initialized) check, before any external calls or state changes. This enforces CEI and prevents reentrancy during initialization.

3.3.6 [L-2] Potential Array Out-of-Bounds Access After Operator Removal

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
PredictionOracle.sol	Business Logic	Low	Low	Addressed

The _distributePayment() function accesses the _operators array using the index _lastOperatorIndex without ensuring that the index remains within bounds after operators are removed. If an operator is removed from the _operators array, _lastOperatorIndex will still point to the previous last element. In this case, _lastOperatorIndex can become equal to or greater than _operators.length, resulting in an out-of-bounds array access (line 729). For example, if the array initially contains three operators and _lastOperatorIndex == 2, removing one operator reduces the array length to two, but _lastOperatorIndex remains 2. Any subsequent access to _operators[2] will revert due to an invalid index. This issue can cause _distributePayment() to revert unexpectedly, potentially disrupting operator fee distribution and leading to denial-of-service conditions for functionality that depends on this logic.

prediction-oracle-contracts-main - PredictionOracle.sol

```
726 function _distributePayment() private {
727     // Send operator fee to next operator (round-robin)
728     if (operatorGasFee > 0 && _operators.length > 0) {
729         address operator = _operators[_lastOperatorIndex];
730         _lastOperatorIndex = uint8(
731             (uint256(_lastOperatorIndex) + 1) % _operators.length
732         );
733         ...
734     }
735     // Accumulate protocol fee
736     accumulatedProtocolFees += protocolFee;
737 }
```

Remediation When removing an operator, if _lastOperatorIndex is greater than or equal to the length of the _operators array, reset _lastOperatorIndex to 0.

3.3.7 [L-3] Array Allocation Fails in PredictionOracle::getPollsByEpochs()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
PredictionOracle.sol	Business Logic	Low	Low	Addressed

The PredictionOracle::getPollsByEpochs() function allocates the return array using a dynamically computed length derived from the _maxResults parameter. When _maxResults is set to zero, the function assigns type(uint256).max as the array size and attempts to allocate new PollInfo[](type(uint256).max). This results in an invalid memory allocation that always reverts, causing the function to fail deterministically.

A similar issue exists in the related function PredictionOracle::getPollsByEpochRange(), which follows the same pattern of treating a zero _maxResults value as “unlimited” and attempts to allocate an unbounded memory array. In both cases, the misuse of type(uint256).max as a sentinel value for unlimited results leads to guaranteed reverts.

prediction-oracle-contracts-main - PredictionOracle.sol

```
449 function getPollsByEpochs(
450     uint32[] calldata _epochs,
451     uint256 _statusFilter,
452     uint256 _typeFilter,
453     uint256 _maxResults
454 ) external view returns (PollInfo[] memory polls) {
455     // If no limit specified, set to max value
456     uint256 maxResults = _maxResults > 0 ? _maxResults : type(uint256).max;
457
458     polls = new PollInfo[](maxResults);
459     uint256 count;
460
461     ...
462 }
```

Remediation Avoid using type(uint256).max as a sentinel value for unlimited results. Instead, enforce a reasonable upper bound on _maxResults to ensure memory allocation is always bounded and cannot revert.

4 Appendix

4.1 About AstraSec

AstraSec is a premier blockchain security firm dedicated to delivering high-quality auditing services for blockchain-based protocols. Composed of veteran security researchers with extensive experience across the DeFi landscape, our team maintains an unwavering commitment to excellence and precision. AstraSec's comprehensive methodology and deep understanding of blockchain architecture enable us to ensure protocol resilience, making us a trusted partner for our clients.

4.2 Disclaimer

The content of this audit report is for informational purposes only and does not constitute legal, financial, or investment advice. The findings, views, and conclusions presented herein are based on the code and documentation provided at the specific time of the audit and may be subject to risks and uncertainties not detected during the assessment.

While AstraSec exerts every effort to ensure the accuracy and completeness of this report, the information is provided on an "as is" basis. We assume no responsibility for any errors, omissions, or inaccuracies. Users should conduct their own independent due diligence and consult with professional advisors before making any investment or deployment decisions. AstraSec shall not be held liable for any direct or indirect losses, damages, or consequences arising from reliance on this report.

4.3 Contact

Phone	+86 156 0639 2692
Email	contact@astrasec.ai
Twitter	https://x.com/AstraSecAI