SECURITY AUDIT REPORT

PlayArts.AI

———

✉ info@0xTeam.Space

✗ 0xTeamSpace

✈ 0xTeamSpace
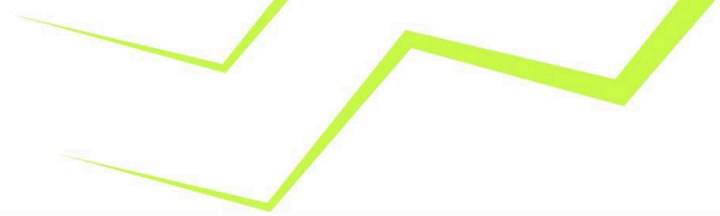
Contents

Revision History & Version Control

Version	Date	Author(s)	Description
1.0	20 January 2025	Bhatia Manoj	Initial Audit Report
2.0	11 February 2025	Bhatia Manoj	Second Audit Report
3.0	14 February 2025	Bhatia Manoj	Final Audit Report

0xTeam conducted a comprehensive Security Audit on the PlayArts.Ai smart contracts to ensure the overall code quality, security, and correctness. The review focused on ensuring that the code functions as intended, identifying potential vulnerabilities, and safeguarding the integrity of PlayArts.Ai's operations against possible attacks.

Report Structure

The report is divided into two primary sections:
1. **Executive Summary** : Provides a high-level overview of the audit findings.
2. **Technical Analysis** : Offers a detailed examination of the smart contract code.

Note :
The analysis is static and manual, exclusively focused on the smart contract code. The information provided in this report should be used to assess the security, quality, and expected behavior of the code.

1.0 Disclaimer

This is a summary of our audit findings based on our analysis, following industry best practices as of the date of this report.However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. The audit focuses on smart contract coding practices and any issues found in the code, as detailed in this report. For a complete understanding of our analysis, you should read the full report. We have made every effort to conduct a thorough analysis, but it's important to note that you should not rely solely on this report and cannot make claims against us based on its contents. We strongly advise you to perform your own independent checks before making any decisions. Please read the disclaimer below for more information.

DISCLAIMER: By reading this report, you agree to the terms outlined in this disclaimer. If you do not agree, please stop reading immediately and delete any copies you have. This report is for informational purposes only and does not constitute investment advice. You should not rely on the report or its content, and 0xTeam and its affiliates (including all associated companies, employees, and representatives) are not responsible for any reliance on this report.The report is provided "as is" without any guarantees. 0xTeam excludes all warranties, conditions, or terms, including those implied by law, regarding quality, fitness for a purpose, and use of reasonable care. Except where prohibited by law, 0xTeam is not liable for any type of loss or damage, including direct, indirect, special, or consequential damages, arising from the use or inability to use this report. The findings are solely based on the smart contract code provided to us.

2.0 Executive Summary

2.1 Overview

0xTeam has meticulously audited the PlayArts.Ai smart contract project from 18 January 2025 to 20 January 2025. The primary objective of this audit was to assess the security, functionality, and reliability of the smart contracts before their deployment on the blockchain. The audit focused on identifying potential vulnerabilities, evaluating the contract's adherence to best practices, and providing recommendations to mitigate any identified risks. The comprehensive analysis conducted during this period ensures that the PlayArts.Ai is robust and secure, offering a reliable environment for its users.

2.2 Scope

The scope of this audit involved a thorough analysis of the PlayArts.Ai Smart Contract, focusing on evaluating its quality, rigorously assessing its security, and carefully verifying the correctness of the code to ensure it functions as intended without any vulnerabilities.

Files in Examination:

Contract Address	0xA7BF2946663309F9A4A31577D96A07aF32f45570
Contracts In-Scope	● Contracts/GotchaWhaleArtwork.sol

OUT-OF-SCOPE: External Solidity smart contract, other imported smart contracts.

2.3 Audit Summary

Name	Verified	Audited	Vulnerabilities
PlayArts.Ai	Yes	Yes	Refer Section 5.0

2.4 Vulnerability Summary

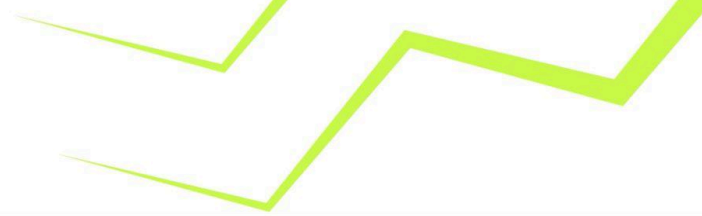| ● High	● Medium	● Low	● Informational
6	6	7	1

● High   ● Medium   ● Low   ● Informational

2.5 Summary of Findings

ID	Title	Severity	Fixed
[H-01]	Bonding Curve's Exponential Growth and Mutable CURVE_SLOPE Lead to Permanent Token Lock Due to Reserve Insufficiency.	HIGH	✔
[H-02]	Precision Loss in Vote Reward Distribution Leads to Incorrect User Payouts	HIGH	✔
[H-03]	DoS vulnerability in finalizeEvent() due to unbounded participant iteration could prevent reward distribution	HIGH	✔
[H-04]	Unbounded Initial Supply Combined with Overflow in Price Calculation Leads to Permanent Price Manipulation	HIGH	✔
[H-05]	getBuyPrice() Function Vulnerable to Arithmetic Overflow Due to Unchecked Multiplication	HIGH	✔
[H-06]	Vote Reward Claims Incorrectly Use ETH Transfer Instead of Configured Payment Token	HIGH	✔
[M-01]	Missing Sequencer Uptime Feed Check for L2 Price Feed Queries	MEDIUM	✔
[M-02]	Precision Loss in Token Creation Fee Calculation Leads to Protocol Fee Rounding	MEDIUM	✔
[M-03]	Insufficient Bounds on Stale Time Threshold	MEDIUM	✔
[M-04]	Arbitrary Vote Price in createVoteEvent	MEDIUM	*
[M-05]	Unvalidated Start and End Timestamps in createVoteEvent	MEDIUM	✔
[M-06]	Potential Mapping Overwrite in Initial721RemixNft's ResultId Storage	MEDIUM	✔
[L-01]	Duplicate Ticker Names Possible Due to Missing Case Sensitivity Check	LOW	✔
[L-02]	Missing Zero Address Check for Payment Token When Using ERC20	LOW	*
[L-03]	Inconsistent Usage of Initial Liquidity Parameter	LOW	✔
[L-04]	Missing Zero Amount Validation in Trading Functions	LOW	✔
[L-05]	Unbounded Vote Event End Time Creates Potential for Indefinite Events	LOW	*
[L-06]	Winner Share Manipulation Risk in Voting System	LOW	*
[L-07]	Unnecessary Receive Function Due to Payable Functions	LOW	✔
[I-01]	Missing Events for Critical Parameter Changes in Owner-Controlled Functions	INFO	*

✔ - Resolved   * - Acknowledged   ✘ - Un Resolved

Severity

	● High	● Medium	● Low	● Informational
Open	6	6	7	1
Resolved	6	5	4	
Acknowledged		1	3	1
Partially Resolved				

(Row label axis: Issues)

- **Open**: Unresolved security vulnerabilities requiring resolution.
- **Resolved**: Previously identified vulnerabilities that have been fixed.
- **Acknowledged**: Identified vulnerabilities noted but not yet resolved.
- **Partially Resolved**: Risks mitigated but not fully resolved.

3.0 Checked Vulnerabilities

We examined the smart contract for widely recognized and specific vulnerabilities. Below are some of the common vulnerabilities considered.

Category	Check Items
Source Code Review	➔ Reentrancy Vulnerabilities ➔ Ownership Control ➔ Time-Based Dependencies ➔ Gas Usage in Loops ➔ Transaction Sequence Dependencies ➔ Style Guide Compliance ➔ EIP Standard Compliance ➔ External Call Verification ➔ Mathematical Checks ➔ Type Safety ➔ Visibility Settings ➔ Deployment Accuracy ➔ Repository Consistency
Functional Testing	➔ Business Logic Validation ➔ Feature Verification ➔ Access Control and Authorization ➔ Escrow Security ➔ Token Supply Management ➔ Asset Protection ➔ User Balance Integrity ➔ Data Reliability ➔ Emergency Shutdown Mechanism

4.0 Techniques , Methods & Tools Used

The following techniques, methods, and tools were used to review all the smart contracts

- **Structural Analysis:**
  EThis involves examining the overall design and architecture of the smart contract. We ensure that the contract is logically organised, scalable, and follows industry best practices. This step is crucial for identifying potential structural issues that could lead to vulnerabilities or maintenance challenges in the future.

- **Static Analysis:**
  Static analysis is conducted using automated tools to scan the contract's codebase for common vulnerabilities and security risks without executing the code. This process helps identify issues such as reentrancy, arithmetic errors, and potential denial-of-service (DOS) vulnerabilities early on, allowing for quick remediation.

- **Code Review / Manual Analysis:**
  A manual, in-depth review of the smart contract's code is performed to verify the logic and ensure it matches the intended functionality as described in the project's documentation. During this phase, we also confirm the findings from the static analysis and check for any additional issues that may not have been detected by automated tools.

- **Dynamic Analysis:**
  Dynamic analysis involves executing the smart contract in various controlled environments to observe its behaviour under different conditions. This step includes running comprehensive test cases, performing unit tests, and monitoring gas consumption to ensure the contract operates efficiently and securely in real-world scenarios.

- **Tools and Platforms Used for Audit:**
  Utilising tools such as Remix , Slither, Aderyn, Solhint for static analysis, and platforms like Hardhat and Foundry for dynamic testing and simulation.

Note: The following values for "Severity" mean:

- High: Direct and severe impact on the funds or the main functionality of the protocol.
- Medium: Indirect impact on the funds or the protocol's functionality.
- Low: Minimal impact on the funds or the protocol's main functionality.
- Informational: Suggestions related to good coding practices and gas efficiency.

5.0 Technical Analysis

High Severity Issues

Issue #1 [Resolved]
[H-1] Bonding Curve's Exponential Growth and Mutable CURVE_SLOPE
Lead to Permanent Token Lock Due to Reserve Insufficiency.

Severity
HIGH

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ `setCurveSlope()` ➔ `getBuyPrice()`

Vulnerability Details
The contract implements a bonding curve where token prices increase exponentially with total supply. The severity of this growth is determined by CURVE_SLOPE, which can be modified by the owner

Vulnerability Code :
https://github.com/0xTeam-Space/01_PlayArts/blob/main/BUGS.md#vulnerability-details

This creates two compounding issues:
1. The exponential growth of prices relative to linear reserve accumulation
2. The owner's ability to increase CURVE_SLOPE, which can accelerate the price growth and potentially trigger immediate reserve insufficiency

Impact
Tokens can become permanently locked due to insufficient reserves.
Owners can modify **CURVE_SLOPE**, accelerating the lockup process.
Exponential growth and mutable slope lead to unpredictable pricing and behavior.
No recovery mechanism exists once reserves are depleted.

Proof of Concept
https://github.com/0xTeam-Space/01_PlayArts/blob/main/BUGS.md#proof-of-concept

Recommended Mitigation Steps

The contract requires fundamental changes to address both the bonding curve and governance issues:

1. Make CURVE_SLOPE immutable

```solidity
contract GotchaWhaleArtWorkV2 is ERC1155URIStorage, Ownable, ReentrancyGuard {
    uint256 public immutable CURVE_SLOPE;

    constructor(uint256 _slope) {
        require(_slope > 0, "Invalid slope");
        CURVE_SLOPE = _slope;
    }
}
```

2. If slope must be mutable, implement strict bounds and timelock

```solidity
uint256 public constant MAX_SLOPE_CHANGE = 10; // 10% max change
uint256 public constant SLOPE_CHANGE_TIMELOCK = 7 days;

function setCurveSlope(uint256 _newSlope) external onlyOwner {
    require(_newSlope <= CURVE_SLOPE * (100 + MAX_SLOPE_CHANGE) / 100, "Exceeds max change");
    require(block.timestamp >= lastSlopeChange + SLOPE_CHANGE_TIMELOCK, "Too soon");
    CURVE_SLOPE = _newSlope;
}
```

3. Implement a reserve ratio requirement that accounts for potential slope changes

```solidity
function calculateMinReserve(uint256 supply, uint256 maxSlope) internal pure returns (uint256) {
    return supply * maxSlope * SAFETY_FACTOR;
}
```

Issue #2 [Resolved]
[H-2] Precision Loss in Vote Reward Distribution Leads to Incorrect User Payouts

Severity:
HIGH

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ `finalizeEvent()`

Description
The contract's vote reward distribution mechanism suffers from precision loss due to integer division rounding. While these funds can be recovered by the contract owner through emergency withdrawal, the core issue is that users receive less than their mathematically correct share of rewards. This becomes especially problematic with high-value native currencies like ETH or BNB, where even small decimal losses translate to significant monetary value.

Impact
Users participating in voting events receive fewer tokens than they mathematically should due to rounding down in integer division. The impact becomes particularly significant when:
1. The reward pool involves expensive native currencies (ETH/BNB)
2. There are many participants with varying vote amounts
3. The total votes don't divide evenly into the winners pool

For example, with ETH at $3000, if a user should receive 1.8 ETH ($5400), they will only get 1 ETH ($3000). The 0.8 ETH difference ($2400) represents a substantial loss in value for the user, occurring due to integer division rather than any intentional fee structure.

Proof of Concept
https://github.com/0xTeam-Space/01_PlayArts/blob/main/BUGS.md#proof-of-concept-1

Recommendation

```
function finalizeEvent(uint256 eventId) external {
    // ... previous code ...

    uint256 baseUnit = 1e18; //@audit -- Use higher precision for calculations

    for (uint256 i = 0; i < voteEvent.participants.length; i++) {
        address participant = voteEvent.participants[i];
        uint256 participantVotes = winnerVotes[participant];
```

```
        if (participantVotes > 0) {
            //@audit -- Calculate reward with higher precision
            uint256 scaledReward = winnersPool * baseUnit;


            uint256 reward = (scaledReward * participantVotes) / totalWinningVotes;
            reward = reward / baseUnit;

            _handleRefund(participant, reward);
        }
    }
}
```

Alternatively, implement percentage-based allocation:

```
// Calculate each participant's percentage with high precision
uint256 percentage = (participantVotes * 1e18) / totalWinningVotes;
uint256 reward = (winnersPool * percentage) / 1e18;
```

References:
1. OpenZeppelin's Math library for safe mathematical operations
2. Similar issue in Uniswap V2's liquidity provider reward distribution
3. EIP-2612: Token Permit for improved precision in token calculations

0xTeam.
W E B 3   A U D I T S

AUDIT REPORT | 12
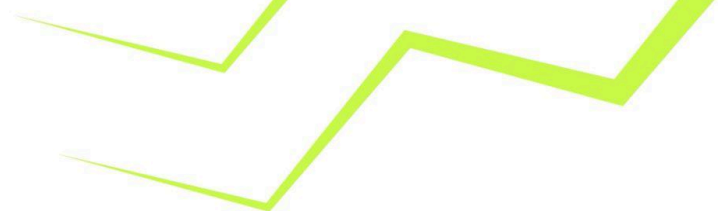
Issue #3 <mark>[Resolved]</mark>
[H-3] DoS vulnerability in finalizeEvent() due to unbounded participant iteration could prevent reward distribution

Severity:
HIGH

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ `finalizeEvent()`

Vulnerability Details
The contract's voting system allows unlimited participants, but the **finalizeEvent()** function iterates through all participants in a single transaction. As participant numbers grow, gas costs increase linearly. Exceeding the block gas limit (~30M on Ethereum) renders the function unexecutable, permanently locking rewards in the contract.

Impact
The finalizeEvent() function contains an unbounded loop that iterates through all participants to distribute rewards. Since there is no upper limit on the number of participants in a voting event, this loop could grow large enough to exceed the block gas limit, making it impossible to distribute rewards and breaking core protocol functionality.

Proof of Concept
https://github.com/0xTeam-Space/01_PlayArts/blob/main/BUGS.md#proof-of-concept-2

Recommendation
```solidity
mapping(uint256 => mapping(address => uint256)) public pendingRewards;

function finalizeEvent(uint256 eventId) external {
    //@audit Calculate and store rewards
    for (uint256 i = 0; i < voteEvent.participants.length; i++) {
        //@audit Store rewards in mapping instead of direct transfer
        pendingRewards[eventId][participant] = reward;
    }
}
function claimReward(uint256 eventId) external {
    uint256 reward = pendingRewards[eventId][msg.sender];
    require(reward > 0, "No reward to claim");
    pendingRewards[eventId][msg.sender] = 0;
    _handleRefund(msg.sender, reward);
}
```
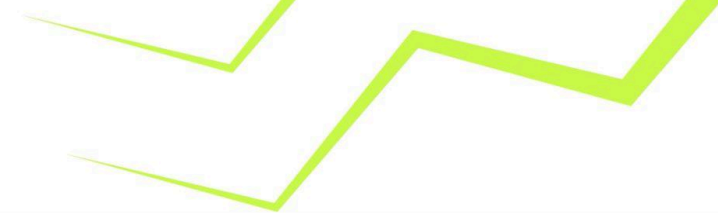
Issue #4 <mark>[Resolved]</mark>
[H-4] Unbounded Initial Supply Combined with Overflow in Price Calculation Leads to Permanent Price Manipulation

Severity:
HIGH

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ `createToken() & getBuyPrice()`

Vulnerability Details
Unbounded initialMintAmount and unchecked arithmetic cause overflows in price calculations when totalSupply is large. This results in permanently incorrect, lower prices, breaking the bonding curve mechanism.

Vulnerability Code :
https://github.com/0xTeam-Space/01_PlayArts/blob/main/BUGS.md#vulnerability-details-2

Impact
Unchecked arithmetic in price calculations causes incorrect pricing once the supply reaches overflow thresholds, leading to two critical issues:

1. **Systemic Price Failure**: Overflow breaks price calculations, permanently returning incorrect (lower) values and disrupting the bonding curve mechanism.
2. **Deliberate Exploitation**: Malicious creators can manipulate the initial supply to trigger overflows, creating predictable arbitrage opportunities. For non-limited tokens, this acts as a permanent exploit, allowing creators to control token pricing unfairly.

Example: At high supply levels, normal price (1000 * CURVE_SLOPE = 1e14) wraps to a smaller value due to overflow, enabling manipulation. Limited tokens are somewhat protected, but non-limited ones remain vulnerable.

Recommendation
1. Add a reasonable maximum cap for initialMintAmount

```
uint256 public constant MAX_INITIAL_SUPPLY = 1e27; // Choose appropriate value
function createToken(...) {
    require(initialMintAmount <= MAX_INITIAL_SUPPLY, "Initial supply too large");
    ...
}
```

2. Remove unchecked block and use safe math for price calculations

Issue #5 <mark>[Resolved]</mark>
[H-5] getBuyPrice() Function Vulnerable to Arithmetic Overflow Due to Unchecked Multiplication

Severity:
HIGH

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ `getBuyPrice()`

Vulnerability Details
In getBuyPrice(), all arithmetic operations are wrapped in an unchecked block, making it susceptible to overflow at multiple points

Vulnerability Code :
https://github.com/0xTeam-Space/01_PlayArts/blob/main/BUGS.md#vulnerability-details-3

Impact
The `getBuyPrice()` function uses unchecked multiplications that can overflow, letting attackers exploit the input amount to mint tokens at much lower prices. Overflows affect both base and curve price calculations, making all tokens vulnerable to this issue.

Proof of Concept
Note The Test case for this Issue is also included in the test/ directory which is the same as for [H-4]. To run the PoC use the command:
```
forge test --via-ir -vv
```

Recommendation
Remove the unchecked block and allow Solidity's default overflow protection to prevent this exploitation:

```solidity
function getBuyPrice(uint256 tokenId, uint256 amount) public view returns (uint256)
{
    TokenInfo storage token = tokens[tokenId];
    uint256 basePart = amount * token.basePrice;  // Will revert on overflow
    uint256 curvePart = (amount * (firstTerm + lastTerm)) / 2;
    return basePart + curvePart;
}
```

Issue #6 [Resolved]
[H-6] Vote Reward Claims Incorrectly Use ETH Transfer Instead of Configured Payment Token

Severity:
HIGH

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ `claimVoteReward()`

Vulnerability Details
The contract supports both native token (ETH) and ERC20 token payments through its PaymentConfig. However, while deposits properly handle both types, the reward distribution always attempts to use ETH transfers:

Vulnerability Code :
https://github.com/0xTeam-Space/01_PlayArts/blob/main/BUGS.md#vulnerability-details-4

When the contract is configured for ERC20 tokens:

1. Users deposit ERC20 tokens through voting
2. Their reward claims attempt ETH transfers instead of ERC20
3. If the contract has no ETH, claims fail entirely
4. If it has ETH, users receive the wrong asset type
5. Original ERC20 deposits remain locked in the contract

The contract already has a proper solution in its _handleRefund function that correctly handles both payment types.

Impact
The mismatch between deposit and withdrawal token types leads to permanent lock of user funds in ERC20-based voting events, since rewards cannot be claimed in the correct token type.

Proof of Concept
Consider this scenario:

1. Contract is configured with USDC as payment token
2. User deposits 1000 USDC by voting
3. Event ends and user attempts to claim reward
4. Claim fails because contract tries to send ETH instead of USDC
5. User's USDC remains permanently locked in contract

Recommendation
Use the existing _handleRefund utility for reward distribution:

```solidity
function claimVoteReward(uint256 eventId) external nonReentrant {
    // ... reward calculation ...
    if (reward > 0) {
        hasClaimedReward[eventId][msg.sender] = true;
        _handleRefund(msg.sender, reward);  // Handles both native and ERC20
        emit RewardClaimed(eventId, msg.sender, reward);
    }
}
```

This ensures users receive rewards in the same token type they deposited.

Medium Severity Issues

Issue #7 [Resolved]
[M-1] Missing Sequencer Uptime Feed Check for L2 Price Feed Queries

Severity
MEDIUM

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ `AggregatorV3Interface` ➔ `getTokenPriceInUSD()`

Description:
The contract uses Chainlink price feeds but lacks sequencer uptime validation for L2 networks like Arbitrum. Without this check, users risk interacting with stale price data if the sequencer goes offline and then resumes operation, as Chainlink feeds may continue serving outdated information.

Impact
If an L2 sequencer goes offline, price feeds may become stale, allowing trades at outdated prices when the sequencer resumes before feeds update. For example, if ETH drops from $2000 to $1800 during downtime, users could briefly trade at the old $2000 price, causing unfair trades and potential losses.

Proof of Concept
https://github.com/0xTeam-Space/01_PlayArts/blob/main/BUGS.md#proof-of-concept-5

Recommendation
Implement L2 sequencer uptime validation before price feed queries:

```
interface iSequencerUptimeFeed {
    function latestRoundData() external view returns (
        uint80 roundId,
        int256 answer,
        uint256 startedAt,
        uint256 updatedAt,
        uint80 answeredInRound
    );
}
```

```solidity
contract GotchaWhaleArtWorkV2 {
    iSequencerUptimeFeed public sequencerUptimeFeed;
    uint256 private constant GRACE_PERIOD_TIME = 3600; // 1 hour or dependent on the
ArtAI business Logic

    constructor(
        // ... other params
        address _sequencerUptimeFeed
    ) {
        sequencerUptimeFeed = iSequencerUptimeFeed(_sequencerUptimeFeed);
    }

    function getTokenPriceInUSD() public view returns (uint256) {
        // Check sequencer status
        (, int256 answer, uint256 startedAt, , ) =
            sequencerUptimeFeed.latestRoundData();

        require(answer == 0, "Sequencer is down");
        require(
            block.timestamp - startedAt > GRACE_PERIOD_TIME,
            "Grace period not over"
        );

        // Existing price feed logic
        // ...
    }
}
```
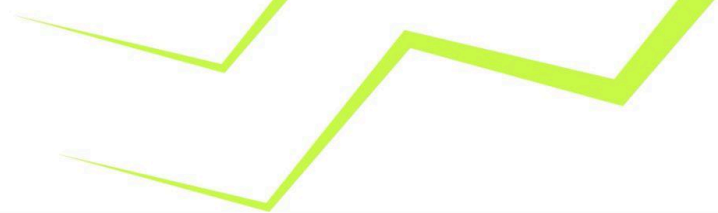
References
https://docs.chain.link/data-feeds/l2-sequencer-feeds

Issue #8 <mark>[Resolved]</mark>
[M-2] Precision Loss in Token Creation Fee Calculation Leads to Protocol Fee Rounding

Severity
MEDIUM

Location	Functions
Contracts/GotchaWhaleArtwork.sol	→ Please Refer Recommendation Section

Vulnerability Details
The fee calculation is performed using the following formula:

```
uint256 requiredFee = (basePrice * fees.creationFee) / 100;
```

Here, fees.creationFee is a percentage value (e.g., 2 for 2%), and the division by 100 is used to convert it to a decimal. However, this direct division can lead to precision loss.
For example: If basePrice is 995 wei and fees.creationFee is 2%:

- Expected fee: 19.9 wei
- Actual fee: (995 * 2) / 100 = 19 wei
- Lost fee: 0.9 wei

While the loss seems small, it grows with larger basePrice values, cumulative token creations, or varying fee percentages set by the owner.

Recommended Mitigation
Implement a fee precision system using basis points (1/10000) instead of percentages:

```
// In state variables
uint256 public constant FEE_PRECISION = 10000;
// In fee calculation
uint256 requiredFee = (basePrice * fees.creationFee) / FEE_PRECISION;
```

And update the fee setting function to work with basis points:

```
function setCreationFee(uint256 _fee) external onlyOwner {
    // 200 would represent 2%
    require(_fee <= 1000, "Fee too high"); // max 10%
    fees.creationFee = _fee; }
```

Issue #9 [Resolved]
[M-3] Insufficient Bounds on Stale Time Threshold

Severity
MEDIUM

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➜ setStaleTimeThreshold()

Impact
The setStaleTimeThreshold function lacks a maximum limit, allowing the owner to set arbitrarily large thresholds that could lead to acceptance of stale price data.
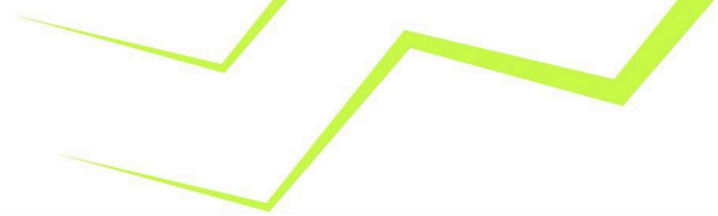
Vulnerability Details

```
function setStaleTimeThreshold(uint256 _threshold) external onlyOwner {
    require(_threshold > 0, "Threshold must be positive");
    staleTimeThreshold = _threshold;
}
```

Recommendation
Add reasonable bounds:

```
uint256 public constant MAX_STALE_THRESHOLD = 1 hours;
require(_threshold <= MAX_STALE_THRESHOLD, "Threshold too high");
```

Issue #10 [Acknowledged]
[M-4] Arbitrary Vote Price in createVoteEvent

Severity
MEDIUM

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➜ `createVoteEvent()`

Description:
The createVoteEvent function does not enforce any constraints on the votePrice parameter. Users can set an arbitrary value, including 0 or excessively high amounts, which can manipulate the voting system.

Impact
1. Exclusive voting pools: Users can set an extremely high vote price, preventing most users from participating.
2. Potential fund mismanagement: The system assumes a minimum price threshold in calculations, which can be bypassed.

Proof of Concept
https://github.com/0xTeam-Space/01_PlayArts/blob/main/BUGS.md#proof-of-concept-poc

Recommendation
1. Implement minimum and maximum vote price constraints.
2. Enforce votePrice to be within a reasonable range (e.g., $1 - $1000 USD equivalent).

OxTeam.
WEB3 AUDITS

Issue #11 [Resolved]
[M-5] Unvalidated Start and End Timestamps in createVoteEvent

Severity
MEDIUM

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ `createVoteEvent()`

Description:
The createVoteEvent function does not enforce any validation rules on the startTime, step1EndTime, step2EndTime, and endTime parameters.
This allows a user to create a voting event with invalid time sequences, such as:

1. Ending before it even starts (endTime < startTime).
2. Step 2 occurring before Step 1 (step2EndTime < step1EndTime).
3. An event with an immediate end time, allowing instant finalization.
4. Never-ending events, preventing proper reward distribution.

Impact
1. Instant event finalization – The event can be finalized immediately upon creation, preventing fair participation.
2. Impossible step progression – The contract logic assumes steps happen sequentially, but users can set timestamps out of order.
3. Denial-of-Service (DoS) – A user can set an unreasonably long duration, locking up funds indefinitely.

Proof of Concept
https://github.com/0xTeam-Space/01_PlayArts/blob/main/BUGS.md#proof-of-concept-poc-1

Recommendation
Ensure that timestamps are logically sequential:

```
require(startTime > block.timestamp && "Start time must be in the future");
require(startTime < step1EndTime, "Step 1 must start after event creation");
require(step1EndTime < step2EndTime, "Step 2 must start after Step 1");
require(step2EndTime < endTime, "End time must be after Step 2");
require(endTime > startTime + 1 days, "End time must be at least 1 day in the future from start time");
```

Introduce a maximum event duration (e.g., 90 days) to prevent indefinite locks

Issue #12 [Resolved]
[M-6] Potential Mapping Overwrite in Initial721RemixNft's ResultId Storage

Severity
MEDIUM

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ `initial721RemixNft()`

Impact
The initial721RemixNft function allows multiple calls with the same resultId, which overwrites previous mappings in resultIdToCreator and resultIdToTokenId. Whether this is intended behavior depends on the business logic, but the code currently allows silent overwrites without any notifications or checks.

Proof of Concept
https://github.com/0xTeam-Space/01_PlayArts/blob/main/BUGS.md#proof-of-concept-poc-2

Recommendation
There are two potential approaches depending on the intended business logic:

1.If resultId should be unique:
```
function initial721RemixNft(...) {
    require(resultIdToTokenId[resultId] == 0, "ResultId already used");
    // Rest of the function
}
```

2.If resultId can be reused but overwrites should be logged:

```
function initial721RemixNft(...) {
    if(resultIdToTokenId[resultId] != 0) {
        emit ResultIdOverwritten(resultId, resultIdToTokenId[resultId],
originalTokenId);
    }
    // Rest of the function
}
```
The development team should clarify the intended behavior of resultId to determine if any changes are needed.

Low Severity Issues

Issue#13 [Resolved]
[L-1]Duplicate Ticker Names Possible Due to Missing Case Sensitivity Check

Severity
LOW

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ createToken()

Vulnerability Details
The createToken function allows creation of tokens with duplicate tickers of different case (e.g., "TEST" and "test"), potentially causing confusion for users and integration issues with other systems.

Impact
This enables the creation of multiple tokens with visually similar tickers that only differ in capitalization, leading to confusion in UI displays, API integrations, and potential market confusion for users trying to trade specific tokens.

Recommendation
```
string memory normalizedTicker = _toLower(ticker);
if (tickerToTokenId[normalizedTicker] != 0) revert TickerAlreadyExists();
```

Issue#14 [Acknowledged]
[L-2]Missing Zero Address Check for Payment Token When Using ERC20

Severity
LOW

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ constructor()

Vulnerability Details
The contract constructor lacks a zero address validation for the payment token

when isNativeToken is false, potentially allowing the contract to be deployed with an invalid ERC20 token address.

Impact

A deployment with an invalid ERC20 address would require contract redeployment and could cause temporary disruption of protocol services. While this is a one-time deployment issue, it could affect protocol credibility.

Recommendation

```
if (!_isNativeToken && _paymentToken == address(0)) revert InvalidPaymentToken();
```

Issue#15 [Resolved]
[L-3]Inconsistent Usage of Initial Liquidity Parameter

Severity
LOW

Location	Functions
Contracts/GotchaWhaleArtwork.sol	→ N/A

Vulnerability Details

The initialLiquidity parameter in token creation is stored but never used, contradicting comments that suggest it should be used for reserves. This creates confusion and wastes gas storing unused data.

Impact

While no funds are at risk, this inconsistency wastes gas storing unused data and creates confusion about the intended functionality of the protocol's liquidity management system.

Recommendation

Either remove the unused parameter or implement the intended reserve functionality.

Issue#16 [Resolved]
[L-4]Missing Zero Amount Validation in Trading Functions

Severity
LOW

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➜ `buyToken()` ➜ `sellToken()`

Vulnerability Details
The buyToken and sellToken functions don't validate that the amount is greater than zero, allowing zero-amount transactions that waste gas and pollute event logs.

Impact
This allows execution of meaningless zero-amount transactions, wasting gas and creating noise in event logs that could complicate protocol analytics and monitoring.

Recommendation
```
require(amount > 0, "Amount must be greater than zero");
```

Issue#17 [Acknowledged]
[L-5]Unbounded Vote Event End Time Creates Potential for Indefinite Events

Severity
LOW

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➜ `createVoteEvent()`

Vulnerability Details
The createVoteEvent function doesn't set a maximum limit for endTime, allowing creation of extremely long-running events that could affect system usability.

Impact

Events could be created with end times far in the future, potentially causing confusion and complicating protocol management. This could also lead to permanently locked funds if the event duration is set extremely high.

Recommendation

```
uint256 public constant MAX_EVENT_DURATION = 30 days;
require(endTime <= startTime + MAX_EVENT_DURATION, "Event too long");
```

Issue#18 [Acknowledged]
[L-6]Winner Share Manipulation Risk in Voting System

Severity
LOW

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➜ NA

Vulnerability Details

The owner can view vote details and manipulate winner shares by adjusting parameters mid-event, potentially compromising vote fairness since all voting data is publicly visible through mappings.

Impact

This transparency combined with mutable parameters creates a risk of owner manipulation, potentially undermining trust in the voting system and its outcomes.

Recommendation
1. Make vote-related parameter changes only apply to new events
2. Add time lock for parameter changes
3. Consider making vote parameters immutable per event

Issue#19 <mark>[Resolved]</mark>
[L-7]Unnecessary Receive Function Due to Payable Functions

Severity
LOW

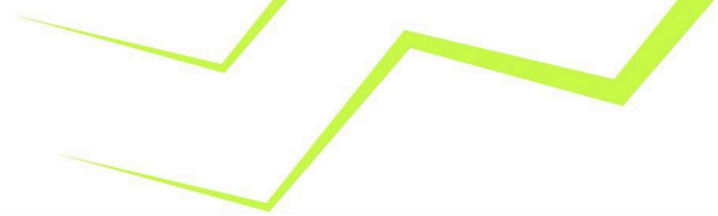| Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ `receive()`

Vulnerability Details
The contract includes a receive() function despite all ETH-accepting functions being marked payable, creating unnecessary complexity and potential confusion.

Impact
The redundant receive function adds unnecessary complexity to the contract and could create confusion about the intended ETH payment flows. While not directly harmful, it violates the principle of minimal implementation.

Recommendation
Remove the receive() function as all intended ETH transfers are handled by payable functions.

Informational Severity Issues

Issue#20 [Acknowledged]
[I-1]Missing Events for Critical Parameter Changes in Owner-Controlled Functions

Severity
INFORMATIONAL

Location	Functions
Contracts/GotchaWhaleArtwork.sol	➔ **Refer Affected functions section below**

Vulnerability Details
Several owner-controlled functions that modify critical protocol parameters don't emit events, making it difficult to track important protocol changes off-chain. While these functions have proper access controls, the lack of events reduces transparency and complicates protocol monitoring.

Impact
The absence of events for parameter changes:

- Makes it harder for users to track important protocol changes
- Complicates off-chain monitoring and indexing
- Reduces protocol transparency
- Makes it difficult to create accurate historical records of parameter changes

Affected functions include:

```solidity
function setEventCreationFee(uint256 _fee) external onlyOwner {
    require(_fee <= MAX_FEE * 100, "Fee exceeds maximum");
    EVENT_CREATION_FEE = _fee;
    // No event emission
}

function setCurveSlope(uint256 _slope) external onlyOwner {
    require(_slope > 0, "Slope must be positive");
    CURVE_SLOPE = _slope;
    // No event emission
}
```

```
function setMaxFee(uint256 _maxFee) external onlyOwner {
    require(_maxFee > 0 && _maxFee <= 100, "Invalid max fee");
    MAX_FEE = _maxFee;
    // No event emission
}

function setMaxCreatorShare(uint256 _share) external onlyOwner {
    require(_share > 0 && _share <= 100, "Invalid creator share");
    MAX_CREATOR_SHARE = _share;
    // No event emission
}
```

Recommendation

Add events for all parameter changes:

```
// Define events
event EventCreationFeeUpdated(uint256 oldFee, uint256 newFee);
event CurveSlopeUpdated(uint256 oldSlope, uint256 newSlope);
event MaxFeeUpdated(uint256 oldMaxFee, uint256 newMaxFee);
event MaxCreatorShareUpdated(uint256 oldShare, uint256 newShare);

// Update functions to emit events
function setEventCreationFee(uint256 _fee) external onlyOwner {

require(_fee <= MAX_FEE * 100, "Fee exceeds maximum");
    uint256 oldFee = EVENT_CREATION_FEE;
    EVENT_CREATION_FEE = _fee;
    emit EventCreationFeeUpdated(oldFee, _fee);
}
```
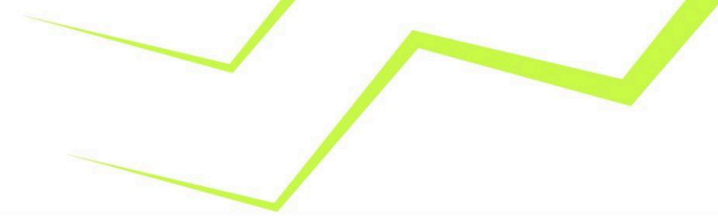
6.0 Auditing Approach and Methodologies Applied

The Solidity smart contract was audited using a comprehensive approach to ensure the highest level of security and reliability. Careful attention was given to the following key areas to ensure the overall quality of the code:

- **Code quality and structure**: We conducted a detailed review of the codebase to identify any potential issues related to code structure, readability, and maintainability. This included analysing the overall architecture of the Solidity smart contract and reviewing the code to ensure it follows best practices and coding standards.
- **Security vulnerabilities**: Our team used manual techniques to identify any potential security vulnerabilities that could be exploited by attackers. This involved a thorough analysis of the code to identify any potential weaknesses, such as buffer overflows, injection vulnerabilities, signatures, and deprecated functions.
- **Documentation and comments**: Our team reviewed the code documentation and comments to ensure they accurately describe the code's intended behaviour and logic. This helps developers to better understand the codebase and make modifications without introducing new issues.
- **Compliance with best practices**: We checked that the code follows best practices and coding standards that are recommended by the Solidity community and industry experts. This ensures that the Solidity smart contract is secure, reliable, and efficient.

Our audit team followed OWASP and Ethereum (Solidity) community security guidelines for this audit. As a result, we were able to identify potential issues and provide recommendations to improve the smart contract's security and performance.

Throughout the audit of the smart contracts, our team placed great emphasis on ensuring the overall quality of the code and the use of industry best practices. We meticulously reviewed the codebase to ensure that it was thoroughly documented and that all comments and logic aligned with the intended behaviour. Our approach to the audit was comprehensive, methodical, and aimed at ensuring that the smart contract was secure, reliable, and optimised for performance.

6.1 Code Review / Manual Analysis

Our team conducted a manual analysis of the Solidity smart contracts to identify new vulnerabilities or to verify vulnerabilities found during static and manual analysis. We carefully analysed every line of code and made sure that all instructions provided during the onboarding phase were followed. Through our manual analysis, we were able to identify potential vulnerabilities that may have been missed by automated tools and ensure that the smart contract was secure and reliable.

6.2 Tools Used for Audit

In the course of our audit, we leveraged a suite of tools to bolster the security and performance of our program. While our team drew on their expertise and industry best practices, we also integrated various tools into our development environment. Noteworthy among them are Remix,Slither, Aderyn, Solhint for Static Analysis and Hardhat & Foundry for Dynamic Analysis. This holistic approach ensures a thorough analysis, uncovering potential issues that automated tools alone might overlook. 0xTeam takes pride in utilising these tools, which significantly contribute to the quality, security, and maintainability of our codebase

7.0 Limitations on Disclosure and Use of this Report

This report contains information concerning potential details of the PlayArts.Ai Project and methods for exploiting them. 0xTeam recommends that special precautions be taken to protect the confidentiality of both this document and the information contained herein. Security Assessment is an uncertain process, based on past experiences, currently available information, and known threats. All information security systems, which by their nature are dependent on human beings, are vulnerable to some degree. Therefore, while 0xTeam considers the major security vulnerabilities of the analysed systems to have been identified, there can be no assurance that any exercise of this nature will identify all possible vulnerabilities or propose exhaustive and operationally viable recommendations to mitigate those exposures. In addition, the analysis set forth herein is based on the technologies and known threats as of the date of this report. As technologies and risks change over time, the vulnerabilities associated with the operation of the PlayArts.Ai Solidity smart contract described in this report, as well as the actions necessary to reduce the exposure to such vulnerabilities, will also change. 0xTeam makes no undertaking to supplement or update this report based on changed circumstances or facts of which 0xTeam becomes aware after the date hereof, absent a specific written agreement to perform the supplemental or updated analysis. This report may recommend that 0xTeam use certain software or hardware products manufactured or maintained by other vendors. 0xTeam bases these recommendations upon its prior experience with the capabilities of those products. Nonetheless, 0xTeam does not and cannot warrant that a particular product will work as advertised by the vendor, nor that it will operate in the manner intended. This report was prepared by 0xTeam for the exclusive benefit of PlayArts.Ai and is proprietary information. The Non-Disclosure Agreement (NDA) in effect between 0xTeam and PlayArts.Ai governs the disclosure of this report to all other parties including product vendors and suppliers.