marchev

Pulse Wars
Security Review

Version 1.2

Reviewed by:

Martin Marchev

Plamen Tsanev

August 6, 2024

Table of Contents

1 Introduction

1.1 About Us

Martin Marchev is an independent security researcher specializing in web3 security. His track record includes top placements in competitive audits such as 1st place in the Immunefi Arbitration contest, 2nd place in the PartyDAO contest (as a team), 3x Top 10 and 5x Top 25 rankings. He is currently ranked #43 on Immunefi's 90-day leaderboard. Martin has responsibly disclosed vulnerabilities in live protocols and has been involved in high-profile audits of projects exceeding $1B in Total Value Locked (TVL), demonstrating his ability to navigate and address complex security challenges.

Plamen Tsanev is a web3 security researcher with numerous top placements in competitive audits, having identified over 40 High/Medium issues. Ranked #5 on Hats Finance's leaderboard, he has led several audits. Plamen has responsibly disclosed vulnerabilities in live protocols, demonstrating a thorough understanding of complex codebases and their vulnerabilities.

1.2 Disclaimer

While striving to deliver the highest quality web3 security services, it is important to understand that the complete security of any protocol cannot be guaranteed. The nature of web3 technologies is such that new vulnerabilities and threats may emerge over time. Consequently, no warranties, expressed or implied, are provided regarding the absolute security of the protocols reviewed. Clients are encouraged to maintain ongoing security practices and monitoring to address potential risks.

1.3 Risk Classification

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

1.3.1 Severity Criteria

- **Critical** - Severe loss of funds, complete compromise of project availability, or significant violation of protocol invariants.
- **High** - High impact on funds, major disruption to project functionality or substantial violation of protocol invariants.
- **Medium** - Moderate risk affecting a noticeable portion of funds, project availability or partial violation of protocol invariants.
- **Low** - Low impact on a small portion of funds (e.g. dust amount), minor disruptions to service or minor violation of protocol invariants.
- **Informational** - Negligible risk with no direct impact on funds, availability or protocol invariants.

2 Executive Summary

2.1 About Pulse Wars

Pulse Wars is an NFT-based strategy game built on the PulseChain blockchain. In the game, players collect, trade, and utilize NFTs to compete in various game modes, which include knockout phases, token trading, and boosting mechanisms. The gameplay combines strategy and chance, requiring players to make tactical decisions to succeed.

2.2 Synopsis

Project Name	Pulse Wars
Project Type	NFT-based Strategy Game
Repository	Pulse-Wars
Commit Hash	543a323c...da0d26e7
Review Period	22 July 2024 - 04 August 2024

2.3 Issues Found

Severity	Count
Critical Risk	2
High Risk	1
Medium Risk	7
Low Risk	7
Informational	3
Total Issues	**20**

3 Findings

Critical Risk

[C-1] Weak randomness source allows gaming various critical aspects of the game

Context: RandomNumber.sol#L25

Description: The NFT game employs a very weak randomness source, making multiple critical aspects of the game vulnerable to exploitation by malicious actors. The weaknesses in the randomness source can be exploited in the following ways:

1. **Minting NFTs of Only a Particular Type**: Malicious actors can precompute random values and determine the type of NFT that will be minted. Given the uniform distribution of the modulo function, the probability for minting any given NFT type is always the same, regardless of rarity. This allows attackers to selectively mint only rare NFTs, severely disrupting the game's economy and rarity mechanics.
2. **Gaming Random Coin Selection**: For NFT types that cannot choose their coins (Power Items, Maxis, Natives, Frenemies & VCs, Pulsicans), attackers can precalculate the set of coins they will receive and only claim the coins if they are satisfied with the selection. This breaks a core invariant of the protocol by allowing these NFT holders to effectively choose their coins.
3. **Manipulating Randomized Coin Selection During Rejoining**: Similar to the above, the rejoining process uses the same weak randomness mechanics, allowing attackers to game the coin selection process.
4. **Exploiting the Boost Mechanism**: Attackers can manipulate the randomness to always achieve the maximum boost percentage of 10%, giving them a consistent and unfair advantage.

The described vulnerabilities can lead to significant economic imbalances, undermine the fairness and competitive integrity of the game, and erode player trust. The ability to predict and manipulate outcomes that should be random creates a critical security issue.

Recommendation: Due to the lack of Chainlink VRF support, which is probably the best solution and industry standard, the next best possible option would be to implement a simplified commit-reveal scheme. When a user requests an NFT, they receive a key (recorded block number). After a certain number of blocks have passed, the user can use their key along with the hash of a future block to mint the actual NFT. This ensures the randomness is secure and unpredictable. The NFT type is determined by combining the user's address and the block hash, providing a fair and tamper-proof mechanism for randomness.

This method secures the randomness by making it dependent on a future, unknown block hash and an opaque user commitment, reducing the risk of manipulation and maintaining the integrity, fairness, and economic balance of the game. Regularly audit and test the randomness implementation to ensure it remains secure against potential vulnerabilities.

Resolution: Partially fixed in `2467076` and `d4adff3`. The Boost mechanism vulnerability has been fixed. The other issues have been acknowledged.

Pulse Wars: Our view is that any player that wishes to seek advantage this way can simply mint a citizen NFT or purchase the class they wish on secondary market. The only NFT that gets any

possible advantage are Founders and the Majority of these have already been minted. Even a player choosing their own class or coins must play the game well to progress.

[C-2] Lack of mechanism to withdraw fees to treasury results in funds irretrievably stuck in the `MainGame` contract

Context: MainGame.sol#L499-L560, MainGame.sol#L568-L630, MainGame.sol#L636-L672

Description: Every round NFT holders are allowed to make trades with their portfolio's coins or "boost" their highest portfolio token by some percentage. All of these operations - buy, sell, boost, rejoin, require some amount of fees paid to the treasury. Their functions are payable and there is validation that the `msg.value` covers those fees. However, only the `rejoin()` function actually sends those fees to the treasury via a direct low-level `call`.

The `buy()`, `sell()` and `boost()` functions never send those fees to the treasury, making them stuck inside the game contract. Inside the contract there is a modifier `deductFees` which is unused, probably intended for exactly this purpose, but it is not implemented on those functions.

Recommendation: Just like in `rejoin()` and the end of the above functions send the received fees to the treasury via a low-level call, or make use of the `deductFees` modifier already defined inside the `MainGame.sol` contract.

Resolution: Fixed in `16ac755`.

High Risk

[H-1] Malicious actor can exploit NFT minting process to mint only specific NFT types

Context: PulseWars.sol#L272

Description: A vulnerability exists in the `PulseWars#mintNFTs()` function where a malicious actor can manipulate the NFT minting process to ensure they only receive NFTs of a particular type. The function uses a random number to determine the NFT type before minting it to the user via the `_safeMint()` function. However, `_safeMint()` uses a callback mechanism to confirm the safe delivery of NFTs. By crafting a malicious contract, an attacker can revert the callback for all unwanted NFT types, thereby guaranteeing the minting of only the desired NFT types.

```
function _safeMint(address to, uint256 tokenId, bytes memory data) internal virtual {
    _mint(to, tokenId);
    ERC721Utils.checkOnERC721Received(_msgSender(), address(0), to, tokenId, data);
    ↪  // @audit This callback can be used to revert in case of unwanted NFT type
}
```

Recommendation: Implement a two-step process for minting NFTs. First, require users to pay for the NFT, which assigns an NFT type based on a random number at the time of payment. This predetermines the NFT type and stores it in the contract. In the second step, users can claim their NFTs. Since the type is locked and the user has already paid, there is no incentive to revert the transaction. This approach ensures the NFT type selection is secure and cannot be manipulated by reverts.

This method secures the NFT minting process by decoupling the payment and type assignment from the actual minting, thus preventing any form of type manipulation by malicious actors.

Resolution: Acknowledged.

Pulse Wars: Our view is that any player that wishes to seek advantage this way can simply mint a citizen NFT or purchase the class they wish on secondary market. The only NFT that gets any possible advantage are Founders and the Majority of these have already been minted. Even a player choosing their own class or coins must play the game well to progress.

Medium Risk

[M-1] No refund of excess ETH when minting Citizens

Context: Citizens.sol#L267-L290

Description: The minting of Citizens, just like PulseWars NFTs, involves paying a minting fee based on a variable threshold, determining if the fee is either low or high. Due to the way transaction ordering works, there are instances in which the minter could accidentally or intentionally(for protection) provide more `msg.value` than needed, which's excess does not refunded.

One such scenario is when the threshold is getting increased while there are pending transactions for minting. Those pending minters would provide the higher `msg.value` pMintingFee. If the transaction increasing the threshold passes first due to gas and ordering, the new threshold could be such that the required amount for the pending minters is actually cMintingFee. With the current configuration, this would make the minters overpay 250_000 with no refund mechanism.

Recommendation: Add a refund mechanism like inside `PulseWars.sol#mintNFTs()`.

Resolution: Acknowledged.

Pulse Wars: Price will be increased to 2M PLS soon so no refunds will be necessary.

[M-2] Improper validation in `updateCategorySupply()` may lead to unlimited minting of NFTs for a given NFT type

Context: PulseWars.sol#L205-L212

Description: The `updateCategorySupply()` function in the `PulseWars` contract does not contain proper validation to ensure that the new category max supply is greater than or equal to `mintedNftCount` for the corresponding `NFTType`. This oversight can lead to serious consequences. If the max supply for a given type is mistakenly set to a value less than the `mintedNftCount`, the following check will never work:

```
while (mintedNftCount[_nftType] == max_supply_per_category[_nftType])
```

This flaw allows an unlimited number of NFTs of the given type to be minted, circumventing the intended supply restrictions.

Example scenario: - The currently minted number of Founder NFTs is 101. - Action: The PulseWars contract owner sets the max supply for the Founder category to 100.

Expected Behavior: No more NFTs of Founder type should be minted.

Actual Behavior: An unlimited number of Founder NFTs can be minted as long as the total supply for the PulseWars contract is not reached.

This can disrupt the economic balance of the game.

Recommendation:

1. Add validation in the `updateCategorySupply()` function to ensure that the new max supply is greater than or equal to the `mintedNftCount` of the corresponding `NFTType`, or
2. Apply the following changes:

```
diff --git a/contracts/core/PulseWars.sol b/contracts/core/PulseWars.sol
index f71b691..fa0b58b 100644
--- a/contracts/core/PulseWars.sol
+++ b/contracts/core/PulseWars.sol
@@ -230,7 +230,7 @@ contract PulseWars is
         NFTType _nftType = NFTType(rn % 6);

         // If the category selected is already full, select the next category
-        while (mintedNftCount[_nftType] == max_supply_per_category[_nftType]) {
+        while (mintedNftCount[_nftType] >= max_supply_per_category[_nftType]) {
             if (uint256(_nftType) < 5) {
                 _nftType = NFTType(uint256(_nftType) + 1);
             } else {
```

These changes will ensure that the max supply constraints are properly enforced, maintaining the intended scarcity of each NFT type.

Resolution: Acknowledged.

Pulse Wars: We will ensure this does not happen.

[M-3] Fully minted NFT type introduces random selection bias in the mint process

Context: PulseWars.sol#L234-L238

Description: Once an NFT type is fully minted, the current implementation introduces a random selection bias in the mint process. The bias arises from the following code:

```
while (mintedNftCount[_nftType] == max_supply_per_category[_nftType]) { //@audit
↪  Unintended random selection bias
    if (uint256(_nftType) < 5) {
        _nftType = NFTType(uint256(_nftType) + 1);
    } else {
        _nftType = NFTType.PULSICANS;
    }
}
```

In this implementation, if a random NFT type is picked and it is fully minted, the next NFT type ID is chosen. This introduces a selection bias: once any of the six types is fully minted, the NFT type with the succeeding ID will have twice the chance of being minted compared to other categories. As

more categories become fully minted, the bias increases, altering the intended uniform distribution and order of NFT minting.

The unintended bias disrupts the intended order of NFT minting and gives certain NFT types a higher chance of being minted once others are fully minted. This affects the fairness and distribution of the NFTs, potentially impacting the game's mechanics and economy.

Recommendation: Use an approach that does not introduce a bias. For example, if an NFT category is fully minted, select a different one using a pseudorandom function such as `keccak256(abi.encodePacked(msg.sender, _rn, block.timestamp))` to ensure unbiased selection among the remaining available categories.

Resolution: Acknowledged.

Pulse Wars: This will only matter with the sale of 5000 more NFTs and can be avoided by minting citizens.

[M-4] Category supplies and total supply can get unsynced

Context: PulseWars.sol#L119-L122, PulseWars.sol#L205-L212

Description: The `max_total_supply` and `max_supply_per_category` are crucial in defining how many NFTs of each category can and should be minted. These values are variable and can be set to different values. However due to lack of proper validation, the 2 variables can accidentally or intentionally get unsynced and cause unwanted behavior when minting new PulseWars tokens. The function that is most impacted is `getRandomSelection()`:

```
function getRandomSelection(uint256 rn) internal view returns (NFTType) {
        require(
            totalSupply() + 1 <= max_total_supply,
            "All NFTs have been minted"
        );
        // random category
        NFTType _nftType = NFTType(rn % 6);

        // If the category selected is already full, select the next category
        while (mintedNftCount[_nftType] == max_supply_per_category[_nftType]) {
            if (uint256(_nftType) < 5) {
                _nftType = NFTType(uint256(_nftType) + 1);
            } else {
                _nftType = NFTType.PULSICANS;
            }
        }
        return _nftType;
    }
```

The invariant is as follows: `SUM(max_supply_per_category) == max_total_supply`. However the 2 setter functions allow 2 scenarios: 1. If the new `max_total_supply` is accidentally/intentionally set to a value greater than the sum of all `max_supply_per_category`, it can create a gas draining DOS, since the while loop above will turn infinite since all categories will be fully minted, but the require check in the beginning of the function will pass 2. If the new `max_total_supply` is accidentally set to a value lower than the sum of all `max_supply_per_category`, then the NFTs will start

getting minted unproportionally. E.g the contract's invariant is that we should have 200 Founders and 1000 Maxis, which is in total 1200. However if due to low funding/low activity, the max supply gets reduced to 1000, then the Founders and Maxis will get minted unproportionally and either one or both categories will be left not fully minted

Recommendation: Both functions `updateCategorySupply()` and `setMaxSupply()` should have proper input validation to ensure they are synced. However there is no merit in having 2 separate setters, so a more elegant approach would be to have a function, for e.g `updateSupplies()` that looks something like:

```
function updateSupplies(uint256 _maxTotalSupply, uint256[6] _categorySupplies) public
↪  onlyOwner {
    require(_maxTotalSupply > totalSupply(), "max value already reached");

    uint256 newCategorySuppliesSum;
    for(uint256 i; i < 6; i++){
        newCategorySuppliesSum += _categorySupplies[i];
    }
    require(_maxTotalSupply == newCategorySuppliesSum, "new supplies aren't synced");

    max_total_supply = _maxSupply;
    for(uint256 i; i < 6; i++){
        max_supply_per_category[NFTType(i)] = _categorySupplies[i];
    }

    //New Events
}
```

Resolution: Acknowledged.

[M-5] Forged `PregameTradeExecuted` event submission

Context: GameLogic.sol#L134-L147

Description: The GameLogic contract's `updateCoinBalanceAfterPregameTrade` is meant to be called internally by the main game whenever a pregame trade gets executed in order to update the corresponding coins' balances and emit an event. However the function itself is marked public. While on it's own it cannot alter critical state variables, it can emit event emissions based on the parameters given to it. This can cause a critical problem to off-chain monitoring systems that rely on the data received by the contract's events for UI, analisys, etc. Even if the protocol currently does not utilize events, any 3rd party app wishing to track the events is subject to being impacted by these fake events.

Recommendation: As the `updateCoinBalanceAfterPregameTrade` is meant to be called only during pregame trading, change it from `public` to `internal`.

Resolution: Fixed in 977508a.

[M-6] Incorrect parameter for Buy orders in both transaction history and the `BuyExecuted` event

Context: MainGame.sol#L620-L628

Description: The Buy transaction history entry as well as the `BuyExecuted` event in the `MainGame` contract incorrectly specifies `_inputAmount` as the amount parameter. `_inputAmount` represents the DAI input amount swapped for the token output amount. The correct parameter should be `_swapValue`, the token output amount from the `_performBuy()` function.

This error leads to misleading information in the transaction history as well as the on-chain events, as it reports the input DAI amount instead of the actual tokens received. This misinformation can affect players who utilize events to analyze competitors and make informed decisions about their trading activity.

Recommendation: Update the Buy transaction history entry and `BuyExecuted` event to use `_swapValue` as the amount parameter instead of `_inputAmount`. This will ensure the transaction history and the event accurately reflect the token output amount, providing correct information for players analyzing trading activities.

Resolution: Fixed in `ae510f2`.

[M-7] The default coin list is fully predictable

Context: Coins.sol#L172-L193

Description: The default coin list is a mechanism for the game for players that missed the initial coin claim period. This mechanism is meant to randomly assign coins to participants who did not initially claim. However, unlike the regular claiming mechanism, the default coin selection is extremely predictable since it is based on a global variable and the id of the NFT getting the default list - both which are parameters that can be known ahead of time. As it can be seen in the context, the random selection is based on the sum of the `defaultRandomNumber` and NFT id. However the `defaultRandomNumber` is a number set only inside `setDefaultSelectionRange` and is the same for every NFT and coin type. This behavior opens up the possibility of players willingly not claiming initially and instead waiting to get their default list, since they can predict if it will yield them a better portfolio, essencially partially gaming their coins.

Recommendation: Get rid of the `defaultRandomNumber` and opt for using the random number contract instead to generate unique values on every call, reducing predictability.

Resolution: Acknowledged.

Pulse Wars: We recognise this but players still need to play very well and get very little benefit in choosing. Plus they will buy more NFTs.

Low Risk

[L-1] `setContractURI()` does not emit `ContractURIUpdated()`

Context: PulseWars.sol#L114

Description: The `PulseWars` contract supports providing contract information via `contractURI()` as per ERC-7572. However, the contract does not emit `ContractURIUpdated()` event when the `contractURI` is updated which is suggested by the ERC spec so that dapps and offchain indexers be notified about the change of the `contractURI`:

> *The `ContractURIUpdated()` event SHOULD be emitted on updates to the contract metadata for offchain indexers to query the contract.*

Recommendation: Emit the `ContractURIUpdated()` event when the `contractURI` is updated.

Resolution: Acknowledged.

[L-2] Multiple tests are failing

Context: N/A

Description: Multiple test cases are failing in the provided repo. Tests are an essential part of the development and security cycle. Consider fixing test cases and ensuring that all tests pass and that the coverage is high.

Recommendation: Fix failing test cases and ensure that all tests pass and provide high code coverage.

Resolution: Partially fixed in `12a915e`. Some tests are still failing.

[L-3] `setMaxSupply()` uses an exclusive > instead of >=

Context: PulseWars.sol#L119-L122, Citizens.sol#L109-L112

Description: The settling of new maximum supply for both NFT contracts serves as to increase caps on the NFTs while the games are running. However in the event we wish to temporarily freeze the supplies without pausing the game, the > check would allow for minting one more NFT, which could be unwanted behavior.

Recommendation: Change the > check to >= to allow stopping minting.

Resolution: Acknowledged.

[L-4] The `getAllBadges()` function is broken due to an off-by-one error in array size calculation leading to array-out-of-bounds error

Context: PulseWars.sol#L349

Description: The size of the result array in the `getAllBadges()` function is incorrectly calculated due to an off-by-one error, which results in an array-out-of-bounds error. For example, when requesting badges from index 0 through 2 (3 elements), the array size is set to 2 instead of 3. Consequently, the function reverts when attempting to assign the third element, as the for loop is inclusive:

```
for (uint256 i = _startIndex; i <= _endIndex; i++) {
  // ...
}
```

This miscalculation leads to a revert and renders the `getAllBadges()` function completely broken.

Recommendation: Correct the `result` array size calculation to account for the inclusive nature of the loop. Ensure the array size is set to `_endIndex - _startIndex + 1` to properly allocate the

required space for all elements in the specified range.

Resolution: Acknowledged.

[L-5] High decimal tokens aren't handled

Context: Coins.sol#L301-L310

Description: The Pulse Wars feature a variety of coins that are virtually simulated, thus any specific ERC20 issues are smartly avoided. However there is a class of tokens with extremely low (<6) and extremely high (>18) decimals. The `getDaiOutputForToken()` function handles everything below 18 decimals correctly, but if the Authorizer decides to add a high decimal token, the calculation would fail.

Recommendation: Handle the case where the token has more decimals than DAI dividing the difference in decimals instead of multiplying, e.g:

```
getDaiOutputForToken(){
...
if(token.decimals > 18) {
    uint256 normalizedTokenAmount = tokenAmount / (10 ** (token.decimals - 18)); //If
    ↪  we have 22 decimals, we reduce them to 18 to normalize for DAI
}
...
```

Resolution: Acknowledged.

Informational

[I-1] Incorrect natspect

Context: PulseWars.sol#L223

Description: The natspec of the `getRandomSelection()` function incorrectly uses the natspec of the `unpauseContract()` function.

Recommendation: Move the current `getRandomSelection()` natspec to the `unpauseContract()` function.

Resolution: Fixed in `44450b3`.

[I-2] Unnecessary max total supply check

Context: PulseWars.sol#L225-L228

Description: The max total supply check in `getRandomSelection()` is unnecessary since the check is already made at the beginning of the `mintNFTs()` function.

Recommendation: Remove the max total supply check in `getRandomSelection()`.

Resolution: Acknowledged.

[I-3] Fix all outstanding TODOs

Context: N/A

Description: The project contains several outstanding TODOs and missing implementations that need to be addressed. These include implementing missing event emissions, finalizing constants, etc. While these issues do not immediately impact the security or core functionality of the contract, they represent incomplete aspects of the project that could lead to potential issues or missed functionality.

Recommendation: Review and address all outstanding TODOs in the project.

Resolution: Fixed in `f5cd370`.

[I-4] Unused `chooseOwnCoin` field in `CategoryConstraint`

Context: MainGame.sol#L32, GameLogic.sol#L82

Description: The `chooseOwnCoin` field in `CategoryConstraint` is intended to indicate whether coins can be selected for a given NFT category and is configured for every NFT category. However, this field is never read or utilized in the contract. Instead, the `selectCoins()` function uses hardcoded logic to determine whether a given NFT category can choose its coins.

Recommendation: Remove the unused `chooseOwnCoin` field or refactor the `GameLogic#selectCoins()` function to use the `chooseOwnCoin` field instead of hardcoded logic.

Resolution: Acknowledged.

[I-5] Unused functions, modifiers and state variables should be removed

Context: Coins.sol#L38-L45, Coins.sol#L312-L331, Coins.sol#L348-L357

Description: The contract contains several unused functions, modifiers and state variables that should be removed to improve readability, reliability, and maintainability:

- The unused state variable `DAI_INFO` is declared but never used.
- The `getTokenAmountForDaiOutput()` function is not used anywhere and is identical to `getTokenOutputForDai()`.
- The `getDaiAmountForTokenOutput()` function is not used anywhere and is identical to `getDaiOutputForToken()`.
- The `deductFees()` modifier is not used anywhere.
- The pause functionalities in `MiniGame` are not used, so the Pausable parent contract should be removed from the list of inherited contracts.

Recommendation: Remove all unused functions, modifiers, and state variables to enhance code clarity and maintainability. This will also help in reducing the potential attack surface and improving the overall reliability of the contracts.

Resolution: Partially fixed in `12dc047`. Some unsused functions and state variables are still present in the code.

[I-6] Various code improvements

Several code improvements have been identified to enhance the efficiency, readability, and maintainability of the contracts:

1. **Unnecessary checks in** `MainGame.sol`**:** There is no scenario where `getAllCoins()` returns an array of 0 coins while `daiBalance` is greater than 0, making these checks redundant. The `daiBalance` is always updated in functions that ensure coins are claimed, guaranteeing consistency. Consider removing the following unnecessary checks.

```
@@ -964,15 +964,6 @@ contract MainGame is Authorizable, Pausable {
                 break;
             }
         }
-        if (hasClaimed) {
-            return hasClaimed;
-        }
-        uint256 _daiBal = gameLogic.daiBalance[_isPW][_nftID];
-        if (_daiBal == 0) {
-            hasClaimed = false;
-        } else {
-            hasClaimed = true;
-        }
         return hasClaimed;
     }
```

2. **Redundant** `if` **statement in** `Coins.sol`**:** The `_tradeCoinType` variable can only be `PULSECHAIN_COINS` or `NON_PULSECHAIN_COINS`, making the `if` statement inside the `else` block superfluous. Simplify the code to remove the unnecessary check.

```
@@ -71,9 +71,7 @@ contract Coins is ICoins, Authorizable {
         if (_tradeCoinType == TradeCoinType.PULSECHAIN_COINS) {
             pulseChainCoinIds.push(totalCoins);
         } else {
-            if (_tradeCoinType == TradeCoinType.NON_PULSECHAIN_COINS) {
-                nonPulseChainCoinIds.push(totalCoins);
-            }
+            nonPulseChainCoinIds.push(totalCoins);
         }
         emit CoinAdded(totalCoins, symbol, _decimals, _priceInDai, _tradeCoinType);
         return totalCoins;
```

3. **Misleading NatSpec in** `MainGame.sol`: The NatSpec comments incorrectly state that fees are denominated in DAI when they are actually in PLS. Update the comments to accurately reflect the fee denomination.

```
@@ -77,11 +77,11 @@ contract MainGame is Authorizable, Pausable {
     /// @dev Instance of the IEligibility interface for eligibility checks.
     IEligibility public eligibilityContract;

-    /// @dev Fee for swapping coins (in DAI).
+    /// @dev Fee for swapping coins (in PLS).
     uint256 public swapFee;
-    /// @dev Fee for boosting a coin (in DAI).
+    /// @dev Fee for boosting a coin (in PLS).
     uint256 public boostFee;
-    /// @dev Fee for rejoining the game (in DAI).
+    /// @dev Fee for rejoining the game (in PLS).
     uint256 public rejoinFee;
     /// @dev Address of the treasury where fees are collected.
     address public treasury;
```

4. **Unnecessary index for Reward Amount in MainGameReward.sol:** Indexing the reward amount is unnecessary and increases gas costs for event emission. Remove the indexing for better gas efficiency:

```
@@ -37,9 +37,9 @@ contract MainGameReward is Pausable, ReentrancyGuard, Authorizable,
 ↪  TimeLock {

     event GameRewardClaimed(uint256 indexed index); //done

-    event GameWinnerDeclared(bool indexed isPW, uint indexed reward,
-                             uint indexed nftID, uint index); //done
+    event GameWinnerDeclared(bool indexed isPW, uint indexed nftID,
+                             uint reward, uint index); //done

-    event GameWinnerEdited(bool indexed isPW, uint indexed reward,
-                           uint indexed nftID, uint index); //done
+    event GameWinnerEdited(bool indexed isPW, uint indexed nftID,
+                           uint reward, uint index); //done
     event EndTimeUpdated(uint256 endTime);

     /**
```

Resolution: Acknowledged.