BeatLand Festival Audit Report

Prepared by Putfor

January 22, 2026

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.

Disclaimer

Putfor 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 research by Putfor 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

I use the CodeHawks `https://docs.codehawks.com/hawks-auditors/how-to-evaluate-a-finding-severity` severity matrix to determine severity. See the documentation for more details.

Audit Details

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

```
5034ccf16e4c0be96de2b91d19c69963ec7e3ee3
```

Scope

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

Compatibilities

Blockchains:
- Ethereum

Tokens:
- Native ETH
- BeatToken is ERC20
- Festival passes and memorabilia are built within the same ERC1155.

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

This security assessment of the `FestivalPass` protocol was conducted using a methodology centered on thorough manual code review combined with targeted, scenario-driven testing. The primary focus of the audit was to analyze core business logic, lifecycle management of memorabilia collections, authorization boundaries, and edge cases arising from state transitions and arithmetic assumptions. Foundry was extensively used to design and execute custom tests that demonstrate exploitability, confirm edge-case behavior, and validate protocol assumptions under realistic conditions.

Automated static analysis tools, including Slither and Aderyn, were also applied as supplementary checks during the review. While these tools did not surface any findings included in the final report, they supported the manual analysis by helping to rule out common vulnerability classes and reinforce confidence in the identified issues.

The audit uncovered several issues across multiple severity levels. Medium-severity findings highlight design and implementation gaps that can lead to permanent misconfiguration of memorabilia collections and unreliable ETH withdrawals under certain conditions. Low-severity issues were also identified, including an off-by-one error in supply enforcement and insufficient input validation in token ID encoding that can trigger unhandled Solidity panic reverts. Collectively, these findings emphasize the importance of robust state management, defensive input validation, and safe ETH transfer patterns to improve protocol reliability, maintainability, and long-term operational safety.

Issues found

Severity	Number of issues found
Medium	2
Low	2
Total	4

Findings

Medium

[M-1] Missing Activation Management for Memorabilia Collections Leads to Permanent Misconfiguration

---

Description

The `MemorabiliaCollection` structure includes an `isActive` flag that determines whether users are allowed to redeem NFTs from a given collection. This flag is set once during collection creation via the `createMemorabiliaCollection` function and is later checked in `redeemMemorabilia` to allow or block redemptions.

However, the protocol does not provide any function to update or toggle the `isActive` flag after a collection has been created. As a result, the activation state of a collection becomes immutable after initialization.

This design leads to inflexible and potentially irreversible behavior: if a collection is created with `isActive = false`, it can never be activated; if it is created with `isActive = true`, it can never be paused or deactivated. The lack of lifecycle management for collections creates a mismatch between the intended use of the `isActive` flag and the actual functionality implemented in the contract.

Risk

Likelihood: Medium

The likelihood is considered medium because this issue stems from normal usage of the protocol. An organizer can unintentionally create a collection with the wrong isActive value, and there is no mechanism to correct it afterward. While this does not require malicious activity, mistakes during collection creation are plausible, especially when handling multiple collections.

Impact: Medium

The impact is medium because a misconfigured collection can either remain permanently inactive, preventing users from redeeming NFTs, or stay perpetually active, preventing the organizer from pausing or deactivating it when needed. This limits operational control and flexibility, potentially causing lost revenue, user frustration, or difficulties in managing the protocol effectively.

Proof of Concept

The following test demonstrates that if a memorabilia collection is created with `isActive` set to `false`, there is no way to activate it later. As a result, the collection becomes permanently unusable and NFTs can never be redeemed from it.

```
function testCannotActivateCollection() public {
    // Organizer creates a collection with isActive set to false
    vm.prank(organizer);
    uint256 collectionId = festivalPass.createMemorabiliaCollection(
        "Test Collection",
        "ipfs://testuri/",
        10e18,
        10,
        false
    );

    // Verify that the collection is indeed inactive after creation
    (,,,,, bool active) = festivalPass.collections(collectionId);
    assert(active == false);

    // A user attempts to redeem an item from the inactive collection
    vm.prank(user1);
    vm.expectRevert("Collection not active");
    festivalPass.redeemMemorabilia(collectionId);
}
```

To run the test, use the following Foundry command:

```
forge test --match-test testCannotActivateCollection
```

Output:

```
Ran 1 test for test/FestivalPass.t.sol:FestivalPassTest
[PASS] testCannotActivateCollection() (gas: 150189)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.64ms (145.28µs
CPU time)
•
Ran 1 test suite in 9.43ms (1.64ms CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)
```

This test confirms that once a collection is created with `isActive = false`, it can never be activated, effectively preventing the collection from ever being redeemed or realized.

Recommended Mitigation

Introduce an explicit function that allows the organizer to update the `isActive` flag of an existing collection. This function should be restricted with the `onlyOrganizer` modifier and enable toggling the collection state between active and inactive, restoring proper lifecycle control over memorabilia collections. For example:

```
+   function setCollectionActive(uint256 collectionId, bool active)
+       external
+       onlyOrganizer
+   {
+       require(collections[collectionId].priceInBeat > 0, "Collection does not
exist");
+       collections[collectionId].isActive = active;
+   }
```

[M-2] Use of `transfer` Causes ETH Withdrawals to Fail for Gas-Heavy Receivers

---

Description

The `withdraw` function transfers ETH using Solidity's built-in `transfer` method:

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

Using `transfer` is considered a bad practice in modern Solidity development. These methods forward a fixed gas stipend of 2300 gas to the recipient. However, due to protocol upgrades such as EIP-1884, EIP-2929, and subsequent gas repricing changes, certain operations that were previously inexpensive — most notably storage writes inside `receive()` or `fallback()` —now require more than 2300 gas.

As a result, if the `target` address is a smart contract whose `receive()` function performs any gas-expensive operation (e.g., modifying storage), the ETH transfer will revert. This makes the withdrawal mechanism unreliable and potentially unusable under valid and realistic conditions.

Risk

Likelihood: High / Impact: Medium

The likelihood is high because the withdrawal function is publicly callable by the owner and does not restrict the `target` address to externally owned accounts. Any attempt to withdraw ETH to a smart contract with a non-trivial `receive()` implementation will consistently revert. The impact is medium: while no funds are directly stolen, ETH held by the contract can become effectively locked, preventing successful withdrawals and breaking a core administrative function of the protocol.

Proof of Concept

The following proof demonstrates that withdrawing ETH fails when the recipient contract requires more than 2300 gas in its `receive()` function.

To reproduce the issue, we first deploy a receiver contract that performs a gas-expensive operation upon receiving ETH. As an example, we modify a storage variable, which exceeds the gas stipend provided by `transfer`.

```
contract GasHeavyReceiver {
    uint256 public x;

    receive() external payable {
        // Storage write intentionally used to exceed the 2300 gas stipend
        x = 1;
    }
}
```

Next, we attempt to withdraw ETH from the protocol contract to this receiver:

```
function testWithdrawWithTransferFails() public {
    // Deploy a receiver contract with a gas-heavy receive() function
    GasHeavyReceiver receiver = new GasHeavyReceiver();

    // Fund the FestivalPass contract with ETH
    vm.deal(address(festivalPass), 1 ether);

    // Attempt withdrawal as the owner
    vm.prank(owner);
    vm.expectRevert();
    festivalPass.withdraw(address(receiver));
}
```

To run the test, use the following Foundry command:

```
forge test --match-test testWithdrawWithTransferFails
```

Output:

```
Ran 1 test for test/FestivalPass.t.sol:FestivalPassTest

[PASS] testWithdrawWithTransferFails() (gas: 91061)

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 11.95ms (1.44ms CPU
time)

Ran 1 test suite in 107.66ms (11.95ms CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)
```

This confirms that under certain realistic conditions, ETH withdrawals become impossible when using `transfer`.

Recommended Mitigation

Replace the use of `transfer` with a low-level `call`, which forwards all remaining gas and properly handles modern gas costs. Additionally, the return value of `call` should be checked to ensure the transfer succeeds.

A corrected version of the withdrawal function would look as follows:

```
function withdraw(address target) external onlyOwner {
-    payable(target).transfer(address(this).balance);
+    (bool success, ) = payable(target).call{value: address(this).balance}("");
+    require(success, "ETH transfer failed");
}
```

Low

[L-1] Off-by-One Error in `maxSupply` Enforcement Leading to Permanent Loss of the Last NFT

Description

The memorabilia redemption logic contains an off-by-one error in the supply cap enforcement.

The contract tracks the next item to be minted using `currentItemId`, which starts at `1` and is incremented after each redemption. However, the redemption guard uses a strict inequality:

```
    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
        _mint(msg.sender, tokenId, 1, "");

        emit MemorabiliaRedeemed(msg.sender, tokenId, collectionId, itemId);
    }
```

This condition prevents minting when `currentItemId` equals `maxSupply`, even though this state corresponds to minting the final valid NFT in the collection. As a result, only `maxSupply - 1` items can ever be redeemed, leaving the last NFT permanently unmintable.

Risk

Likelihood: High

The issue is deterministic and will occur for every memorabilia collection regardless of external conditions or user behavior. Any collection created with `maxSupply > 0` will consistently suffer from this off-by-one error, making the likelihood of occurrence high.

Impact: Low

The impact is limited but tangible: one NFT per collection becomes permanently inaccessible. This results in an incorrect supply cap, broken collection economics, and potential trust issues for organizers and users expecting the advertised maximum supply to be fully mintable.

Proof of Concept

The following test demonstrates that a collection with `maxSupply = 3` allows only two successful redemptions, while the third redemption attempt always reverts.

```
function test_MaxSupply_OffByOne() public {
    // Organizer creates a collection with a declared maxSupply of 3
    vm.startPrank(organizer);
    uint256 collectionId = festivalPass.createMemorabiliaCollection(
        "Test Collection",
        "ipfs://test",
        1e18,
        3,      // maxSupply = 3
        true
    );
    vm.stopPrank();

    // Fund user with enough BEAT tokens to redeem all items
    vm.prank(address(festivalPass));
    beatToken.mint(user1, 10e18);

    // First redemption: itemId = 1 (allowed)
    vm.prank(user1);
    festivalPass.redeemMemorabilia(collectionId);

    // Second redemption: itemId = 2 (allowed)
    vm.prank(user1);
    festivalPass.redeemMemorabilia(collectionId);

    // Third redemption should mint itemId = 3,
    // but instead reverts due to the off-by-one supply check
    vm.prank(user1);
    vm.expectRevert("Collection sold out");
    festivalPass.redeemMemorabilia(collectionId);
}
```

The test can be executed using Foundry:

```
forge test --match-test test_MaxSupply_OffByOne
```

Output:

```
Ran 1 test for test/FestivalPass.t.sol:FestivalPassTest
[PASS] test_MaxSupply_OffByOne() (gas: 339325)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.18ms (1.37ms CPU
time)

Ran 1 test suite in 82.06ms (9.18ms CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)
```

As demonstrated, the final NFT in the collection can never be redeemed, effectively reducing the usable supply by one.

Recommended Mitigation

Align the supply check with the chosen indexing scheme.

The simplest and safest fix is to allow minting while the number of minted items is strictly less than `maxSupply`. Given that `currentItemId` starts at `1`, the condition should be updated to:

```
    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");


        // 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
        _mint(msg.sender, tokenId, 1, "");

        emit MemorabiliaRedeemed(msg.sender, tokenId, collectionId, itemId);
}
```

This change preserves the existing item ID scheme while ensuring that exactly `maxSupply` NFTs can be minted, restoring correct supply enforcement without requiring structural changes to the contract.

[L-2] Unsafe Token ID Encoding Can Trigger Unhandled Solidity Panic Reverts

---

Description

The protocol implements a custom token ID encoding scheme that combines a `collectionId` and an `itemId` into a single `uint256` value by packing each value into 128 bits:

```
function encodeTokenId(uint256 collectionId, uint256 itemId) public pure returns
(uint256) {
    return (collectionId << COLLECTION_ID_SHIFT) + itemId;
}
```

Decoding is performed by shifting and casting operations that assume both values fit within a 128-bit range:

```
function decodeTokenId(uint256 tokenId) public pure returns (uint256
collectionId, uint256 itemId) {
    collectionId = tokenId >> COLLECTION_ID_SHIFT;
    itemId = uint256(uint128(tokenId));
}
```

While this design implicitly assumes that `collectionId` and `itemId` are bounded to 128 bits, the functions accept unconstrained `uint256` inputs and do not perform any explicit validation.

If a caller provides a value larger than `type(uint128).max`, the left-shift operation in `encodeTokenId` will overflow and trigger a Solidity panic revert (`0x11`). Unlike a standard `require`-based revert, a panic indicates a low-level arithmetic error and is not gracefully handled or explained by the contract logic.

Although this behavior prevents silent data corruption, it introduces an unhandled failure mode where malformed inputs cause abrupt execution termination instead of a controlled, descriptive revert. This makes the function more brittle and harder to safely integrate or extend.

Risk

Likelihood: Low

The likelihood of this issue being triggered is low under normal protocol operation, as current usage patterns are expected to rely on well-formed, internally generated identifiers that remain within safe bounds. However, since `encodeTokenId` is a public helper function accepting arbitrary `uint256` values, it may be called with invalid inputs in edge cases, future extensions, or external integrations. The absence of explicit input validation increases the chance of unexpected panics during such usage.

Impact: Low

The impact is low, as the issue does not enable asset theft, data corruption, or invariant violations. Its primary effect is reduced robustness: callers may encounter hard panics (`0x11`) instead of clean, developer-friendly error messages. This can complicate debugging, testing, and composability, especially when the function is reused in broader systems or user-facing tooling.

Proof of Concept

The following test demonstrates that passing oversized input values to `encodeTokenId` results in a Solidity panic rather than a controlled revert.

```
// This test intentionally triggers a Solidity Panic (0x11),
// not a standard revert with an error message.
function testIncorrectData() public {
    uint256 oversized = type(uint128).max + 1;

    // This call will revert with Panic(0x11) due to an invalid left shift.
    // Note: Panic reverts cannot be caught using vm.expectRevert(),
    // which further highlights the lack of graceful error handling.
    festivalPass.encodeTokenId(oversized, 1);
}
```

This test is expected to fail with a panic error. The presence of `Panic(0x11)` confirms that invalid inputs are not validated and instead cause a low-level arithmetic failure, demonstrating the issue described above.

Recommended Mitigation

Add explicit input validation in `encodeTokenId` to ensure that both `collectionId` and `itemId` fit within 128 bits before performing bit-shift operations. This prevents panic reverts and replaces them with clear, intentional error handling.

```
function encodeTokenId(uint256 collectionId, uint256 itemId) public pure returns
(uint256) {
+    require(collectionId <= type(uint128).max, "collectionId out of range");
+    require(itemId <= type(uint128).max, "itemId out of range");
     return (collectionId << COLLECTION_ID_SHIFT) + itemId;
}
```

This change preserves the intended encoding scheme while ensuring that invalid inputs are rejected safely and transparently.