2025-07-20-beatland-festival

Beatland Festival Audit Report

Prepared by Demaxl

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

The Demaxl 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 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

Issues found

Category	No. of Issues
High	3
Medium	1
Low	0

Findings

High

[H-1] Reseting the current pass supply to `0` in the `FestivalPass::configurePass` function allows users to bypass the max supply cap of a pass

Description:

```
function configurePass(uint256 passId, uint256 price, uint256 maxSupply) external onlyOrganizer {
        require(passId == GENERAL_PASS || passId == VIP_PASS || passId == BACKSTAGE_PASS, "Invalid pass ID");
        require(price > 0, "Price must be greater than 0");
        require(maxSupply > 0, "Max supply must be greater than 0");

        passPrice[passId] = price;
        passMaxSupply[passId] = maxSupply;

@>      passSupply[passId] = 0; // Reset current supply
    }
```

If you reset `passSupply[passId]` to `0` in the `FestivalPass::configurePass` function after passes have been sold, the next buyer will be able to mint as if no passes have been sold.

This allows the total minted passes to exceed `passMaxSupply`, which is a serious vulnerability (a supply cap bypass)

Impact:

- Supply caps become meaningless: The users can mint unlimited passes beyond the intended maximum supply
- Pass scarcity and value are destroyed, affecting the economic model

Proof of Concept:

```
 function test_SupplyCapBypassVulnerability() public {
        // Step 1: Configure a pass with max supply of 2
        vm.prank(organizer);
        festivalPass.configurePass(1, GENERAL_PRICE, 2);

        // Step 2: Buy 2 passes (reaching max supply)
        vm.prank(user1);
        festivalPass.buyPass{value: GENERAL_PRICE}(1);

        vm.prank(user2);
        festivalPass.buyPass{value: GENERAL_PRICE}(1);

        // Verify max supply reached
        assertEq(festivalPass.passSupply(1), 2);
        assertEq(festivalPass.passMaxSupply(1), 2);

        // Step 3: Try to buy another pass - should fail
        address user3 = makeAddr("user3");
        vm.deal(user3, 10 ether);
        vm.prank(user3);
        vm.expectRevert("Max supply reached");
        festivalPass.buyPass{value: GENERAL_PRICE}(1);

        // Step 4: VULNERABILITY - Organizer reconfigures the pass
        // This resets passSupply[1] to 0, bypassing the supply cap!
        vm.prank(organizer);
        festivalPass.configurePass(1, GENERAL_PRICE, 2);

        // Step 5: Now we can buy more passes even though max supply was already
reached
        vm.prank(user3);
        festivalPass.buyPass{value: GENERAL_PRICE}(1);

        // Step 6: We can even buy more passes beyond the original max supply
        vm.deal(user4, 10 ether);
        vm.prank(user4);
        festivalPass.buyPass{value: GENERAL_PRICE}(1);

        // Step 7: Verify the vulnerability - total supply exceeds max supply
        assertEq(festivalPass.passSupply(1), 2); // Current supply counter
        assertEq(festivalPass.passMaxSupply(1), 2); // Max supply limit

        // But we actually have 4 passes minted in total!
        assertEq(festivalPass.balanceOf(user1, 1), 1);
        assertEq(festivalPass.balanceOf(user2, 1), 1);
        assertEq(festivalPass.balanceOf(user3, 1), 1);
        assertEq(festivalPass.balanceOf(user4, 1), 1);

        // Total minted: 4 passes, but max supply is only 2!
        uint256 totalMinted = festivalPass.balanceOf(user1, 1) + festivalPass.balanceOf(user2, 1)
            + festivalPass.balanceOf(user3, 1) + festivalPass.balanceOf(user4, 1);

        assertGt(totalMinted, festivalPass.passMaxSupply(1), "VULNERABILITY: Total minted exceeds max supply!");
    }
```

Recommended Mitigation:

The `passSupply` reset should be removed

```
function configurePass(uint256 passId, uint256 price, uint256 maxSupply) external onlyOrganizer {
    require(passId == GENERAL_PASS || passId == VIP_PASS || passId == BACKSTAGE_PASS, "Invalid pass ID");
    require(price > 0, "Price must be greater than 0");
    require(maxSupply > 0, "Max supply must be greater than 0");

    passPrice[passId] = price;
    passMaxSupply[passId] = maxSupply;

-    passSupply[passId] = 0;
}
```

[H-2] `FestivalPass::getMultiplier` logic flaw allows unlimited reward exploitation

Description:

The `FestivalPass::getMultiplier` function uses a simple if-else hierarchy that only checks for the presence of pass types, not their quantity or intended usage.

```
// Get user's reward multiplier based on pass type
    function getMultiplier(address user) public view returns (uint256) {

        if (balanceOf(user, BACKSTAGE_PASS) > 0) {
            return 3; // 3x for BACKSTAGE
        } else if (balanceOf(user, VIP_PASS) > 0) {
            return 2; // 2x for VIP
        } else if (balanceOf(user, GENERAL_PASS) > 0) {
            return 1; // 1x for GENERAL
        }
        return 0; // No pass
    }
```

For example, once a user owns just 1 backstage pass, they permanently receive the `3x` multiplier for all future performances, regardless of how many other passes they own or how many times they attend performances.

Impact:

Users can game the system by purchasing just one backstage pass to unlock permanent 3x multipliers

Proof of Concept:

```
 function test_MultiplierBugVulnerability() public {
        // Step 1: Create some performances for testing
        uint256 baseReward = 100e18;

        vm.startPrank(organizer);
        uint256 performanceId1 = festivalPass.createPerformance(
            block.timestamp + 1 hours, // Start in 1 hour
            10 hours, // Duration
            baseReward // Base reward: 100 BEAT getTokenBalances
        );
        uint256 performanceId2 = festivalPass.createPerformance(
            block.timestamp + 1 hours, // Start in 1 hour
            10 hours, // Duration
            baseReward // Base reward: 100 BEAT tokens
        );
        uint256 performanceId3 = festivalPass.createPerformance(
            block.timestamp + 1 hours, // Start in 1 hour
            10 hours, // Duration
            baseReward // Base reward: 100 BEAT tokens
        );
        vm.stopPrank();

        // Step 2: User buys 1 backstage pass (just to get the 3x multiplier) and 2 general passes
        vm.startPrank(user1);
        festivalPass.buyPass{value: GENERAL_PRICE}(1);
        festivalPass.buyPass{value: GENERAL_PRICE}(1);

        festivalPass.buyPass{value: BACKSTAGE_PRICE}(3); // 1 Backstage pass
        vm.stopPrank();

        // Step 3: VULNERABILITY - Check multiplier logic
        // The bug: getMultiplier() only checks if user has ANY backstage pass
        // It doesn't consider the quantity or other pass types
        uint256 multiplier = festivalPass.getMultiplier(user1);
        assertEq(multiplier, 3, "User gets 3x multiplier with just 1 backstage pass");
        // Step 4: Demonstrate the economic impact
        // Fast forward to performance time
        vm.warp(block.timestamp + 1 hours + 30 minutes);

        // User attends all performances and gets 3x multiplier for each even though they only have 1 backstage pass
        vm.startPrank(user1);
        festivalPass.attendPerformance(performanceId1);
        vm.warp(block.timestamp + COOLDOWN);
        festivalPass.attendPerformance(performanceId2);
        vm.warp(block.timestamp + COOLDOWN);
        festivalPass.attendPerformance(performanceId3);
        vm.stopPrank();

        // Check BEAT token balance

        // 3x multiplier for each of the three performances
        uint256 actualPerformanceReward = baseReward * 3 ∗ 3; // 900e18
        uint256 actualTotal = BACKSTAGE_WELCOME_BONUS + actualPerformanceReward;

        //  "User received backstage welcome bonus + 3x multiplier reward for the three performances
        assertEq(beatToken.balanceOf(user1), actualTotal);

        // Step 5: Demonstrate the unfair advantage

        // The user has 2 general passes and 1 backstage pass
        // so they should have (1 * baseReward) + (1 * baseReward) + (3 * baseReward) = 500e18
        // after using all their passes to attend the performances
        uint256 fairPerformanceReward = (1 * baseReward) + (1 * baseReward) + (3 * baseReward);
        uint256 fairTotal = BACKSTAGE_WELCOME_BONUS + fairPerformanceReward;

        // The user is able to cheat the system just buying 1 backstage pass and forever benefitting from its 3x multiplier
        assertGt(actualTotal, fairTotal);
    }
```

Recommended Mitigation:

Track which passes have been used for specific performances

[H-3] Pass Reuse Vulnerability Allows Unlimited Performance Attendance

Description

The `FestivalPass::attendPerformance` contract contains a critical vulnerability where festival passes are not burned after use. This allows users to attend multiple performances with a single pass and get the BeatToken rewards

In the `FestivalPass::attendPerformance` function:

```
// 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);
        BeatToken(beatToken).mint(msg.sender, performances[performanceId].baseReward * multiplier);

        emit Attended(msg.sender, performanceId, performances[performanceId].baseReward * multiplier);



    }
```

The function only tracks attendance per performance (`hasAttended[performanceId][msg.sender]`) but doesn't consume the pass, allowing unlimited reuse across different performances.

Impact

- Users can earn significantly more BEAT tokens than intended by attending multiple performances with a single pass purchase
- The festival loses potential revenue as users don't need to buy additional passes for multiple performances
- Excessive BEAT token minting due to unlimited performance attendance causes token inflation

Proof of Concept

Consider this scenario:

1. A user buys 1 VIP pass for 0.1 ETH (should only allow 1 performance)
2. User attends 4 performances, earning 800 BEAT tokens instead of 200 BEAT tokens
3. **Unfair advantage: 600 BEAT tokens (3x more than intended)**

Proof of Code

```
function test_PassReuseVulnerability() public {
    // Step 1: Create multiple performances for testing
    uint256 baseReward = 100e18;

    vm.startPrank(organizer);
    uint256 performanceId1 = festivalPass.createPerformance(
      block.timestamp + 1 hours, // Start in 1 hour
      2 hours, // Duration
      baseReward // Base reward: 100 BEAT tokens
    );
    uint256 performanceId2 = festivalPass.createPerformance(
      block.timestamp + 4 hours, // Start in 4 hours
      2 hours, // Duration
      baseReward // Base reward: 100 BEAT tokens
    );
    uint256 performanceId3 = festivalPass.createPerformance(
            block.timestamp + 7 hours, // Start in 7 hours
            2 hours, // Duration
            baseReward // Base reward: 100 BEAT tokens
        );
        vm.stopPrank();

        // Step 2: User buys only 1 VIP pass
        vm.startPrank(user1);
        festivalPass.buyPass{value: VIP_PRICE}(2); // 1 VIP pass
        vm.stopPrank();

        // Step 3: VULNERABILITY - User attends all three performances with the sam
e pass
        // Fast forward to first performance
        vm.warp(block.timestamp + 1 hours + 30 minutes);

        vm.startPrank(user1);
        festivalPass.attendPerformance(performanceId1);
        vm.stopPrank();

        // Fast forward past cooldown and to second performance
        vm.warp(block.timestamp + COOLDOWN + 2 hours + 30 minutes);

        vm.startPrank(user1);
        festivalPass.attendPerformance(performanceId2);
        vm.stopPrank();

        // Fast forward past cooldown and to third performance
        vm.warp(block.timestamp + COOLDOWN + 2 hours + 30 minutes);

        vm.startPrank(user1);
        festivalPass.attendPerformance(performanceId3);
        vm.stopPrank();

        // Step 4: Verify the vulnerability - user still has their pass
        // The pass should have been burned after each use, but it wasn't!
        assertEq(
            festivalPass.balanceOf(user1, 2),
            1,
            "VULNERABILITY: User still has their pass after attending 3 performances!"
        );

        // Step 5: Calculate the unfair rewards
        uint256 vipMultiplier = 2; // VIP gets 2x multiplier
        uint256 totalRewardEarned = baseReward * vipMultiplier * 3; // 2x multiplier for 3 performances
        uint256 totalWithBonus = VIP_WELCOME_BONUS + totalRewardEarned;

        // User received welcome bonus + 2x multiplier for all three performances
        assertEq(beatToken.balanceOf(user1), totalWithBonus, "User earned rewards for all 3 performances");

        // Step 6: Demonstrate the economic impact
        // Fair scenario: User should only be able to attend 1 performance with 1 pass
        uint256 fairReward = VIP_WELCOME_BONUS + (baseReward * vipMultiplier); // Only 1 performance
        uint256 unfairAdvantage = totalWithBonus - fairReward;

        assertGt(totalWithBonus, fairReward, "VULNERABILITY: User earned more than they should have!");
        assertEq(unfairAdvantage, baseReward * vipMultiplier * 2, "User got 2 extra performance rewards");

        // Step 7: Show that the user can even attend more performances
        // Create another performance
        vm.startPrank(organizer);
        uint256 performanceId4 = festivalPass.createPerformance(
            block.timestamp + 2 hours, // Start in 2 hours
            2 hours, // Duration
            baseReward // Base reward: 100 BEAT tokens
        );
        vm.stopPrank();

        // Fast forward to the new performance
        vm.warp(block.timestamp + 2 hours + 30 minutes);
        vm.startPrank(user1);
        festivalPass.attendPerformance(performanceId4);
        vm.stopPrank();

        // User can attend a 4th performance with the same pass!
        assertEq(
            festivalPass.balanceOf(user1, 2), 1, "VULNERABILITY: User can attend unlimited performances with 1 pass!"
        );

        // Final reward calculation
        uint256 finalReward = VIP_WELCOME_BONUS + (baseReward * vipMultiplier * 4); // 4 performances
        assertEq(beatToken.balanceOf(user1), finalReward, "User earned rewards for 4 performances with 1 pass");

        console.log("VULNERABILITY DEMONSTRATED:");
        console.log("User bought 1 VIP pass for", VIP_PRICE, "ETH");
        console.log("User attended 4 performances and earned", finalReward, "BEAT tokens");
        console.log("Fair reward should have been", VIP_WELCOME_BONUS + (baseReward * vipMultiplier), "BEAT tokens");
        console.log(
            "Unfair advantage:", finalReward - (VIP_WELCOME_BONUS + (baseReward * vipMultiplier)), "BEAT tokens"
        );
    }
```

Recommended Mitigation

A pass is represented as a token in the `FestivalPass` contract (ERC 1155) so we burn the pass after use

Medium

[M-1] Collections created with `activateNow = false` cannot be activated later, causing permanent denial

# of service

Description

The `FestivalPass::createMemorabiliaCollection` function allows organizers to create collections with an `activateNow` parameter that determines whether the collection is immediately active for redemption. However, there is no mechanism to activate collections that were created with `activateNow = false.`
When a collection is created, the
`isActive` field is set based on the `activateNow` parameter:

```
collections[collectionId] = MemorabiliaCollection({
    name: name,
    baseUri: baseUri,
    priceInBeat: priceInBeat,
    maxSupply: maxSupply,
    currentItemId: 1,
@>    isActive: activateNow
});
```

The `FestivalPass::redeemMemorabilia` function requires collections to be active:

```
require(collection.isActive, "Collection not active");
```

Since there's no function to modify the `isActive` state after creation, collections created with `activateNow = false` become permanently unusable.

Impact

Collections created with `activateNow = false` are permanently locked and cannot be used for their intended purpose. This creates a denial of service for organizers who intended to activate these collections later. The organizer must create new collections to achieve the same functionality, leading to wasted gas and potential confusion.

Proof of Concept

1. Organizer calls `createMemorabiliaCollection` with `activateNow = false`:

```
festivalPass.createMemorabiliaCollection("Future Collection", "ipfs://QmXXX", 100
e18, 10, false);
```

1. Collection is created with `isActive = false`
2. Users attempt to redeem from the collection:

   ```
   festivalPass.redeemMemorabilia(collectionId); // Reverts with "Collection not active"
   ```

3. No function exists to activate the collection, making it permanently unusable

Recommended Mitigation

Add a function to allow organizers to activate collections after creation:

```
function activateCollection(uint256 collectionId) external onlyOrganizer {
    require(collections[collectionId].priceInBeat > 0, "Collection does not exist");
    require(!collections[collectionId].isActive, "Collection already active");
    collections[collectionId].isActive = true;
    emit CollectionActivated(collectionId);
}
```

Additionally, consider adding a corresponding `deactivateCollection` function for complete control over collection states.