Mythos Studios

Audit

Presented by:

OtterSec contact@osec.io

Akash Gurugunti sud0u53r.ak@osec.io

Vishvesh Rao vishvesh@osec.io

Contents

01 | Executive Summary

Overview

Mythos Studios engaged OtterSec to perform an assessment of the `mythosairdrop.sol` program. This assessment was conducted between April 19th and April 20th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 1 finding total.

We provided recommendations regarding `sendTokensBatch` to potentially optimize gas usage (OS-MYT-SUG-00).

Overall we commend the Mythos Studios team for their attention to detail and responsiveness.

Scope

The source code was delivered to us in a git repository github.com/westcoastnft/mythos-contracts/blob/main/contracts/MythosAirdrop.sol. This audit was performed against commit 9ab04b2.

A brief description of the programs is as follows.

Name	**Description**
MythosAirdrop.sol	The contract provides features for conducting an airdrop of ERC721A tokens. Tokens can be sent individually to a recipient address or grouped together and sent in bulk to multiple recipients in a single transaction.

02 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-MYT-SUG-00	Gas optimization recommendations regarding `sendTokensBatch`.

OS-MYT-SUG-00 | Gas Optimization

Description

In `MythosAirdrop.sol`, `sendTokensBatch` may be optimized to reduce gas usage. While the current `MAX_SUPPLY` value of 100 may not benefit from gas optimization, batching tens of thousands of tokens may.

The suggested gas optimizations are:

1. Unchecked math for `i++` in the loop.
2. Duplicate `supplyAvailable` modifier invocations due to its place as an internal function.
3. Inline internal functions to avoid unnecessary calls

MythosAirdrop.sol SOLIDITY

```solidity
/**
 * @notice sends tokens to a batch of addresses.
 * @param receivers array of addresses to send tokens.
 */
function sendTokensBatch(ReceiverData[] calldata receivers)
    external
    onlyOwner
    nonReentrant
{
    uint256 receiversLength = receivers.length;
    for (uint256 i; i < receiversLength) {
        ReceiverData receiver = receivers[i]

        address to = receiver.to;
        uint32 numberOfTokens = receiver.numberOfTokens;

        if (_totalMinted() + numberOfTokens > MAX_SUPPLY)
        revert ExceedsMaximumSupply();

        _safeMint(to, numberOfTokens);

        unchecked{++i};

    }
}
```

Remediation

Integrate the recommendations provided to optimize gas usage.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority or access control validation • Improperly designed economic incentives leading to loss of funds
High	Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.