SALUS SECURITY

DEC 2024

CODE SECURITY ASSESSMENT

YALA

Overview

Project Summary

- Name: Yala - Contract_Core
- Platform: EVM-compatible chains
- Language: Solidity
- Repository:
    - https://github.com/yalaorg/yala-contract-core
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	Yala - Contract_Core
Version	v2
Type	Solidity
Dates	Jan 02 2025
Logs	Dec 04 2024; Jan 02 2025

Vulnerability Summary

Total High-Severity issues	0
Total Medium-Severity issues	0
Total Low-Severity issues	1
Total informational issues	2
Total	3

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Inaccurate totalMinted counter	Low	Business Logic	Resolved
2	Missing two-step transfer ownership pattern	Informational	Business Logic	Resolved
3	Gas optimization suggestions	Informational	Gas Optimization	Resolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Inaccurate totalMinted counter

Severity: Low	Category: Business Logic
Target: - contracts/NFT/YetiNFT.soll	

Description

The contract's `totalMinted` counter increments whenever a new NFT is minted but does not decrement when an NFT is burned. This leads to a discrepancy between `totalMinted` and the actual number of NFTs in existence. Here's the scenario illustrating the issue:

1. Initial State: `totalMinted = 0`.
2. Mint an NFT to a user: `totalMinted = 1`.
3. The user burns the NFT, but `totalMinted` remains at 1.
4. The contract owner mints another NFT with the same tokenId. Since the `_tokenExists` function checks `ownerOf(tokenId)` and gets `address(0)`, it returns false, allowing the minting process to proceed.
5. `totalMinted` increments to 2, even though only one NFT exists.

If this process repeats, `totalMinted` could erroneously reach the `MAX_SUPPLY` limit, preventing legitimate minting of new NFTs.

Recommendation

Avoid reusing burned/minted `tokenIds`.

Status

The team has resolved this issue in commit f71ccce.

2.3 Informational Findings

2. Missing two-step transfer ownership pattern

Severity: Informational	Category: Business Logic
Target: - contracts/NFT/YetiNFT.sol	

Description

The Yetiasd contract uses a custom function `changeOwner` which is a simple mechanism to transfer the ownership not supporting a two-step transfer ownership pattern. This simpler mechanism can be useful for quick tests, but projects with production concerns are likely to outgrow it. Transferring ownership is a critical operation and this could lead to transferring it to an inaccessible wallet or renouncing the ownership, e.g. mistakenly.

contracts/NFT/YetiNFT.sol:L55-L57

```
function changeOwner(address newOwner) public onlyOwner{
    owner =  newOwner;
}
```

Recommendation

It is recommended to implement a two-step transfer of ownership mechanism where the ownership is transferred and later claimed by a new owner to confirm the whole process and prevent lockout.

Status

The team has resolved this issue in commit f71ccce.

3. Gas optimization suggestions

Severity: Informational	Category: Gas Optimization

Target:
- contracts/NFT/YetiNFT.sol

Description

1.Memory reading saves more gas than storage reading multiple times when the state is not changed. So caching the storage variables in memory and using the memory instead of storage reading is effective. Cache array length outside of the loop can save gas.

contracts/NFT/YetiNFT.sol:L29

```
for (uint256 i = 0; i < recipients.length; i++) {
```

2.The `_mintNFTToAddress` has checked if `tokenIds` exists. There is a duplicate check in the `airdrops` function.

contracts/NFT/YetiNFT.sol:L26-L40

```
function airdrops(address[] calldata recipients, uint[] calldata tokenIds) external
onlyOwner{
    require(totalMinted + recipients.length <= MAX_SUPPLY, "Airdrop exceeds max supply");
    require(recipients.length == tokenIds.length,"Recipients and tokenIds length
mismatc");
    for (uint256 i = 0; i < recipients.length; i++) {
        require(!_tokenExists(tokenIds[i]), "Token ID already minted");
        _mintNFTToAddress(recipients[i], tokenIds[i]);
    }
}

function _mintNFTToAddress(address recipient, uint256 tokenId) internal {
    require(totalMinted < MAX_SUPPLY, "Max supply reached");
    require(!_tokenExists(tokenId), "Token ID already exists");
    _safeMint(recipient, tokenId, "");
    totalMinted++;
}
```

Recommendation

Consider using the above suggestions to save gas.

Status

The team has resolved this issue in commit f71ccce.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit be20f50:

File	SHA-1 hash
contracts/NFT/YetiNFT.sol	a100b70e105e73cac3a45650b5ae698f8905b37c