QuillAudits

# Audit Report
# May, 2022

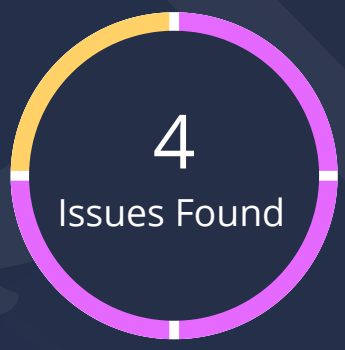
For

ChainCollection

# Table of Content

# Executive Summary

| | |
|---|---|
| **Project Name** | ChainCollection-closedseaNFT-standardRoyalty |
| **Overview** | Chain Collection is The First ZERO-FEES Multi-Chain NFT, NFT Games and Metaverse Marketplace |
| **Timeline** | April 25th, 2022 to May 9th, 2022 |
| **Method** | Manual Review, Functional Testing, Automated Testing etc. |
| **Scope of Audit** | The scope of this audit was to analyse ChainCollection codebase for quality, security, and correctness. |
| **Sourcecode** | *https://github.com/adilghani/chaincollection-contracts/tree/main* |
| **Commit** | d095337ddac3043139fc1e275a15400d6c713207 |
| **Fixed in** | *https://github.com/adilghani/chaincollection-contracts/commit/f4af6c258d93ce36386df213bf6196da8613f7f1* |
| **Commit** | f4af6c258d93ce36386df213bf6196da8613f7f1 |




High
Medium
Low
Informational

| | High | Medium | Low | Informational |
|---|---|---|---|---|
| Open Issues | 0 | 0 | 0 | 0 |
| Acknowledged Issues | 0 | 0 | 0 | 1 |
| Partially Resolved Issues | 0 | 0 | 0 | 0 |
| Resolved Issues | 0 | 1 | 0 | 2 |

## Types of Severities

### High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

### Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

### Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

### Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

## Types of Issues

### Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

### Resolved

These are the issues identified in the initial audit and have been successfully fixed.

### Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

### Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

# Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility leve

# Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

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

### Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

### Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

### Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

### Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

### Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

# Manual Analysis

## High Severity Issues

No issues were found

## Medium Severity Issues

### A.1 Function Fails to return dust

**2171**

```
function mintWithETH(string memory _uri,uint256 royaltyValue) external payable {
  uint seaBal = sea.balanceOf(msg.sender);
  if (seaBal < seaAmountForExemptFee) {
    require(msg.value >= mintPriceInETH, 'ClosedSeaNFT: mint price insufficient');
    feeAddress.transfer(msg.value);
  }
```

**Description**

mintWithETH() function fails for test cases with msg.value above mintPriceInETH and seaBal of msg.sender is less than seaAmountForExemptFee.
This is because

**line: 3016** feeAddress.transfer(msg.value) - transfers all the ether to the feeAddress, after which again on

**line: 3031** payable(msg.sender).transfer(dust) - the contract tries to return dust that is “msg.value - mintPriceInEth”. But, this is not possible as complete msg.value was already sent to feeAddress and there is no ETH left in the transaction to return.

Thus, the transaction always fails for this scenario.
For example, the test case with seaAmountForExemptFee = 5000, seaBal = 0, mintPriceInEth = 1 ether and msg.value = 2 ether will fail.

**Remediation**

To fix this, change line: #3016 to “feeAddress.transfer(mintPriceInETH);”
After doing this, only the mintPriceInETH will be forwarded to the feeAddress and the rest will get returned to the msg.sender.

**Status**

**Fixed**

## Low Severity Issues

No issues were found

## Informational Issues

### A.2 Unlocked pragma (pragma solidity ^0.8.0)

**Description**

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.
In case of this contract, certain functionalities have been used that only came into existence after the version 0.8.4 and we have been informed that the contract was tested on compiler version 0.8.7, but in the contract itself 0.8.0's unlocked version is used.

**Remediation**

Here all the in-scope contracts have an unlocked pragma, it is recommended to lock the same (0.8.7). Moreover, we strongly suggest not to use experimental Solidity features (e.g., pragma experimental ABIEncoderV2) or third-party unaudited libraries. If necessary, refactor the current code base to only use stable features.

**Status**

**Fixed**

## A.3 Mint count initialization

**2961**

```
uint256 public tokenCountMinted = 1;
```

**Description**

Token mint count variable is initialized with one that could cause error while counting the total number of minted NFTs.

**Recommendation**

The variable should be initialized with zero.

**Status**

**Fixed**

## A.4 General Recommendation

In the light of recent events, we conclude in our audit that certain libraries such as EnumerableMap and EnumerableSet are known to consume a lot of gas. We suggest to use them carefully.

Imported contracts (by OpenZeppelin) are using solidity version till 0.8.0 and the main contract was tested with the functionalities of version 0.8.7 and it is not recommended to use these versions while deployment with locked pragmas. We recommend using latest imports by OpenZeppelin.

**Status**

**Acknowledged**

# Functional Testing

**Some of the tests performed are mentioned below**

- should be able to mint with ETH and return dust
- Should be able to set token royalty
- Should be able to set amount for exempt fee
- Should be able to set sea token address
- Should be able to set mint price and fee address
- Should revert if mint price is insufficient

# Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

# Closing Summary

In this report, we have considered the security of the ChainCollection. We performed our audit according to the procedure described above.

Some issues of Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture. In the End,Chain Collection Team Resolved all Issues.

# Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the ChainCollection Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the ChainCollection team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

# About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

**500+**
Audits Completed

**$15B**
Secured

**500K**
Lines of Code Audited

## Follow Our Journey

# Audit Report
# May, 2022

For

ChainCollection

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com