QuillAudits

# Audit Report
# May, 2022

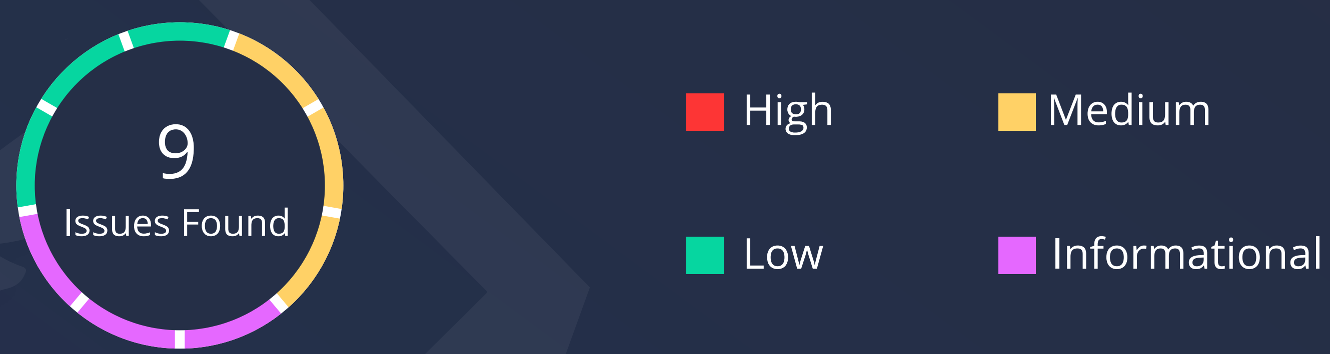
For

ChainCollection

# Table of Content

# Executive Summary

| | |
|---|---|
| **Project Name** | ChainCollection -NFTController |
| **Overview** | The First ZERO-FEES Multi-Chain NFT, NFT Games and Metaverse Marketplace |
| **Timeline** | April 25th, 2022 to 16 May, 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. |
| **Source code** | *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 |
|---|---|---|---|---|
| Open Issues | 0 | 0 | 0 | 0 |
| Acknowledged Issues | 0 | 0 | 1 | 1 |
| Partially Resolved Issues | 0 | 0 | 0 | 0 |
| Resolved Issues | 0 | 3 | 2 | 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
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded

Severus.finance - Audit Report

- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

# 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 Testing

## High Severity Issues

No issues found

## Medium Severity Issues

### 1. Function should be internal

**Line #1858**

```
function readyToSellTokenTo(
    address _tokenAddr,
    uint256 _tokenId,
    uint256 _price,
    // uint256 _royalty,
    address _to,
    string memory _category,
    bool _withEther
) public whenNotPaused notBanned(_tokenAddr, _tokenId)
```

**Description**

This function can be called by anyone as it is public and the address parameter “_to” can be set to any arbitrary address instead of the address of the msg.sender. Making it public would lead to readyToSellTokenTo() being called directly and the address _to parameter can be set arbitrarily.

**Remediation**

readyToSellTokenTo() function should be made internal and not public as readyToSellTokenTo() is being used internally in readyToSellToken().

**Status**

**Fixed**

## 2. Missing zero check and same address check

**Line #1764**

```
constructor(
    address _nftAddress,
    address _quoteErc20Address,
    address payable _feeAddr,
    uint256 _feePercent,
    address _weth,
    address _usdToken
) public {
    require(_nftAddress != address(0) && _nftAddress != address(this));
    require(_quoteErc20Address != address(0) && _quoteErc20Address != address(this));
    require(_feePercent <= MAX_FEE_PERCENT);
    closedSeaNFT = IclosedSeaNft(_nftAddress);
    quoteErc20 = IERC20(_quoteErc20Address);
    feeAddr = _feeAddr;
    feePercent = _feePercent;
    weth = IWETH(_weth);
    usdToken = IERC20(_usdToken);
    _operators[_msgSender()] = true;
    emit FeeAddressTransferred(address(0), feeAddr);
    emit SetFeePercent(_msgSender(), 0, feePercent);
}
```

### Description

There is missing zero check for both “usdToken” and “weth” addresses as well as same address checks. According to the functionality of the contract, these addresses can only be set once (during deployement) and if it’s set to zero or any non-existent address then there could be loss of funds. Moreover, these addresses can also be set as the same which may cause some functions to fail.

### Recommendation

Consider adding zero address and same address checks to avoid this issue.

### Status

**Fixed**

## 3. Missing zero check

**Line #1764 and #2088**

```
function transferFeeAddress(address payable _feeAddr) public {
    require(_msgSender() == feeAddr, 'FORBIDDEN');
    feeAddr = _feeAddr;
    emit FeeAddressTransferred(_msgSender(), feeAddr);
}
```

### Description

Missing Zero check on FEEADDR_ in the constructor and the logic to update it afterwards states that only the feeAddress owner can call the function to change it.
In a scenario where FeeAddress is set to zero address or a Non-existing address will lead to lost of this functionality completely as well as the funds in the form of fee will be lost

### Remediation

Consider adding zero address check in the constructor and instead of using a require check in the transferFeeAddr function, a modifier should be used.

### Status

**Fixed**

## Low Severity Issues

### 4. Improper Implementation

**Line #2171**

```
    uint256 len = _tokenBids[key].length;
    for (uint256 i = _index; i < len - 1; i++) {
        _tokenBids[key][i] = _tokenBids[key][i + 1];
    }
    _tokenBids[key].pop();
}
```

**Description**

The implementation of this logic results is that the index should be less than length. Also, the cases of non-existing indexes should be handled.

**Remediation**

If the index doesn't exist, the function should not return any number and there should be checks to handle the non-existing indexes.

**Status**

**Fixed**

## 5. Issue with fee deduction

**Line #1811, 1818**

```
if (_tokenAskedWithEther[key]) {
    require(msg.value >= price, 'pay amount insufficient');
        if (feeAmount != 0 && userSeaTokenBalance < seaAmountForExemptFee) { /
        feeAddr.transfer(feeAmount); // 2% to feeAddr
        }
        // Royalty Implementation
        if(recipient != closedSeaNFT.ownerOf(_tokenId)){
            // distribute royalty to recipient from 98%
            payable(recipient).transfer(royaltyAmount); // transfer Royalty to
            payable(seller).transfer(price.sub(feeAmount.add(royaltyAmount)));
        }
```

### Description

If an user is using Ether and their sea token balance is greater than the sea amount for exempt fee but the feeAmount is not equal to zero then the condition on line 1811 will return false but there is a possibility that the recipient is not the owner of the tokenId, and in that case, fee amount will still be deducted while transferring the token.
Hence, the logic of the contract which states that if an user has a certain amount of SeaTokenBalance will not have to pay the fee will fail.

### Remediation

Consider checking the sea token balance again in the condition on the line “#1815”.

### Status

**Fixed**

## 6. Insufficient Tests Provided

### Description

There was insufficient test coverage of the codebase provided to us When such a critical project does not provide all the details about what to expect from and functions and logic, it is not recommended from security perspective

### Recommendation

It is advised that the team cover at least 80 percent of the test cases

### Status

**Acknowledged**

## Informational Issues

### 7. 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.

**Recommendation**

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**

### 8. Multiple Pragmas used (pragma solidity ^0.6.0 <0.8.0)

**Description**

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.

**Recommendation**

Use latest imports by OpenZeppelin

**Status**

**Fixed**

### 9. General Recommendation

**Description**

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.

**Status**

**Acknowledged**

# Functional Testing

**LiquidityGeneratorToken.sol**

- ✓ Should be able to buy and sell token
- ✓ Should be able to bid, update the bidding price and cancel bid
- ✓ Should be able to cancel sale of token
- ✓ Should be able to set fee address and rate
- ✓ Should be able to set price for token
- ✓ Should be able to ban and release token
- ✓ Should be able to pause and unpause functionalities
- ✓ Should revert if WETH and USD token are either same or the zero address
- ✓ Should revert if fee address is a zero address
- ✓ Should revert if seller is not owner
- ✓ Should revert if owner tries to bid
- ✓ Should revert if the Token is not in sell book
- ✓ Should revert if the bidder tries to buy without cancelling the bid

# 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 Medium, 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, ChainCollection 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