

# D'Agora NFT & Marketplace

## Smart Contract Audit Report Prepared for D'Agora



---

<b>Date Issued:</b>	Aug 21, 2023
<b>Project ID:</b>	AUDIT2022025
<b>Version:</b>	v2.0
<b>Confidentiality Level:</b>	Public



## Report Information

Project ID	AUDIT2022025
Version	v2.0
Client	DAgora
Project	DAgora NFT & Marketplace
Auditor(s)	Patipon Suwanbol Darunphop Pengkumta Sorawish Laovakul
Author(s)	Darunphop Pengkumta Sorawish Laovakul
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

## Version History

Version	Date	Description	Author(s)
2.0	Aug 21, 2023	Update issue	Wachirawit Kanpanluk
1.0	Apr 21, 2022	Full report	Darunphop Pengkumta Sorawish Laovakul

## Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	<a href="https://t.me/inspexco">t.me/inspexco</a>
Email	<a href="mailto:audit@inspex.co">audit@inspex.co</a>

---

# Table of Contents

<b>1. Executive Summary</b>	<b>1</b>
1.1. Audit Result	1
1.2. Disclaimer	1
<b>2. Project Overview</b>	<b>2</b>
2.1. Project Introduction	2
2.2. Scope	3
<b>3. Methodology</b>	<b>4</b>
3.1. Test Categories	4
3.2. Audit Items	5
3.3. Risk Rating	7
<b>4. Summary of Findings</b>	<b>8</b>
<b>5. Detailed Findings Information</b>	<b>10</b>
5.1. Transaction Ordering Dependence	10
5.2. Centralized Control of State Variable	14
5.3. Unchecked Return Value of ERC20 transferFrom()	16
5.4. Unbound Configuration Parameter	19
5.5. Incorrect Order of Operations	22
5.6. Smart Contract with Unpublished Source Code	25
5.7. Insufficient Logging for Privileged Functions	26
5.8. Improper Function Visibility	28
5.9. Use of Duplicate Literals	31
5.10. Inexplicit Solidity Compiler Version	34
5.11. Unnecessary Function due to Duplicate Functionality	35
<b>6. Appendix</b>	<b>37</b>
6.1. About Inspex	37

## 1. Executive Summary

As requested by DAgora, Inspex team conducted an audit to verify the security posture of the DAgora NFT & Marketplace smart contracts between Apr 7, 2022 and Apr 12, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of DAgora NFT & Marketplace smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

### 1.1. Audit Result

In the initial audit, Inspex found 2 high, 1 medium, 3 low, 1 very low, and 4 info-severity issues. With the project team's prompt response 2 high, 1 medium, 2 low, 1 very low, and 4 info-severity issues were resolved or mitigated in the reassessment, while 1 low-severity issues were acknowledged by the team. Therefore, Inspex trusts that DAgora NFT & Marketplace smart contracts have high-level protections in place to be safe from most attacks.



### 1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

## 2. Project Overview

### 2.1. Project Introduction

Dagora is the leading Multichain NFT Marketplace. It allows anyone to buy, sell, and auction NFTs on Ethereum Mainnet, BNB Smart Chain, Avalanche, Fantom, HECO Chain, Polygon, Arbitrum and Boba Network. The user can switch from one blockchain to another in just a few clicks.

DagoraMarketplace is an NFT marketplace. The platform applies the signature mechanism to help the sellers list their NFTs with a zero gas cost. Additionally, the NFTs creators can gain the royalty fee of the collection by registering to the DagoraMarketplace contract and they will receive the DAGoraCollection NFTs as the owner representative.

#### Scope Information:

Project Name	DAgora NFT & Marketplace
Website	<a href="https://dagora.xyz/">https://dagora.xyz/</a>
Smart Contract Type	Ethereum Smart Contract
Chain	Ethereum, BNB Smart Chain, Avalanche C-Chain, Fantom Opera, HECO, Polygon, Arbitrum, Boba Network
Programming Language	Solidity
Category	NFT, Marketplace

#### Audit Information:

Audit Method	Whitebox
Audit Date	Apr 7, 2022 - Apr 12, 2022
Reassessment Date	Apr 20, 2022

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox:** The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox:** Only the bytecodes of the smart contracts are provided for the assessment.

## 2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

### Initial Audit

Contract	Bytecode SHA256 Hash
DagoraCollection	9aab282d8008369998d9a8c9aef0618b8d952a6689402ca320a7fd8c1746ecf6
DagoraMarketplace	a21b4b429af1620d593c3d28e6399d5fa2b1ce44bc65d6178a3ac3c7b4148c1f

### Reassessment

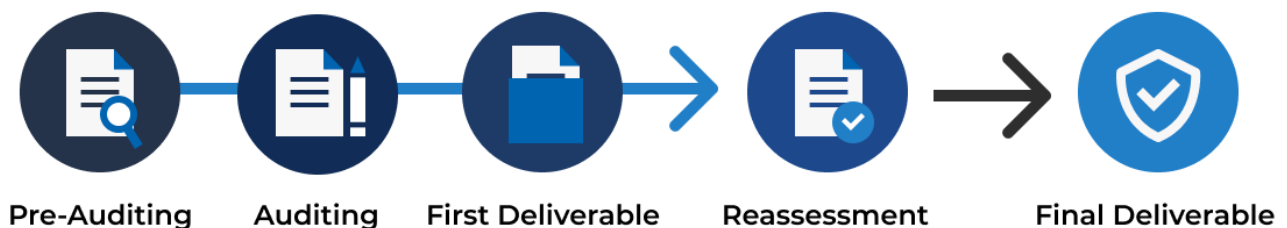
Contract	Bytecode SHA256 Hash
DagoraCollection	cd95efa12b5cff0b7c165581ee6df447ca2fd219cf583404106f996e40e071bb
DagoraMarketplace	50ace70da5a01df503aa0a7d000def69b8a2722b12ae951a02bb723ec20a19c7

As the DAGora team has decided not to publish the source code to protect their intellectual property, the users should compare the bytecode hashes with the smart contracts compiled with solidity version 0.7.6 before interacting with them to make sure that they are the same with the contracts audited.

## 3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing:** Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing:** Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting:** Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment:** Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable:** Providing a full report with the detailed status of each issue



### 3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

## 3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Smart Contract with Unpublished Source Code
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Insufficient Logging for Privileged Functions
Invoking of Unreliable Smart Contract
Use of Upgradable Contract Design
Centralized Control of State Variable
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control

Broken Authentication
Improper Kill-Switch Mechanism
Improper Front-end Integration
Insecure Smart Contract Initiation
Denial of Service
Improper Oracle Usage
Memory Corruption
<b>Best Practice</b>
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

### 3.3. Risk Rating

OWASP Risk Rating Methodology ([https://owasp.org/www-community/OWASP\\_Risk\\_Rating\\_Methodology](https://owasp.org/www-community/OWASP_Risk_Rating_Methodology)) is used to determine the severity of each issue with the following criteria:

- **Likelihood:** a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact:** a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

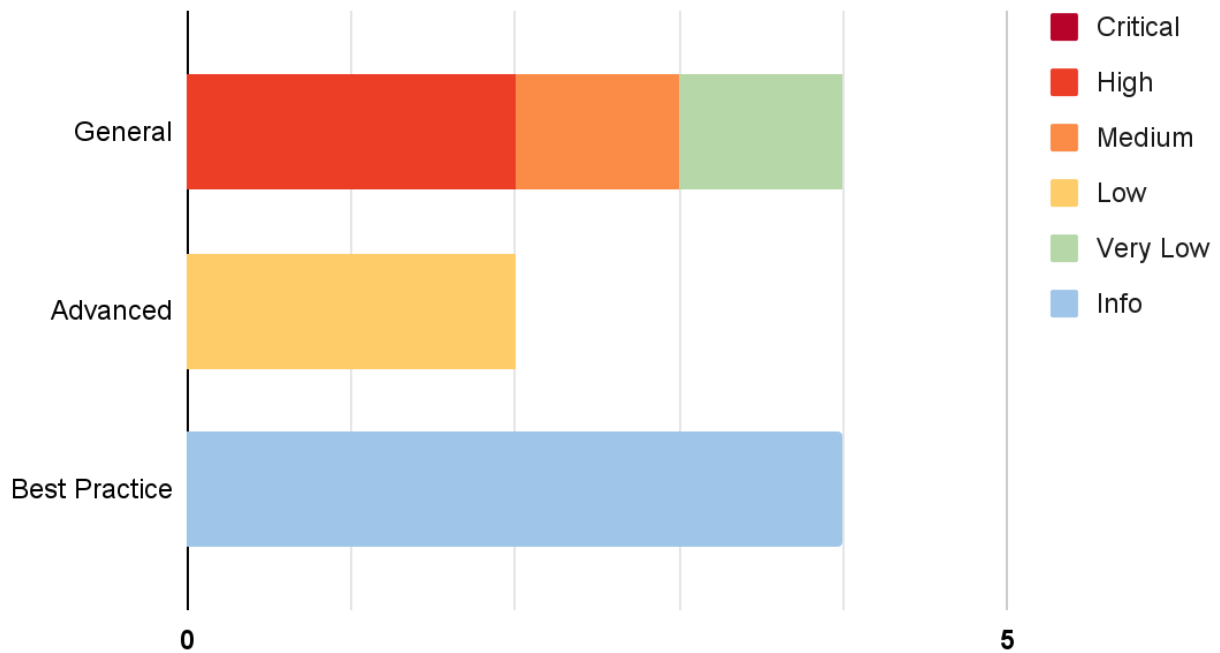
**Severity** is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Likelihood		
	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

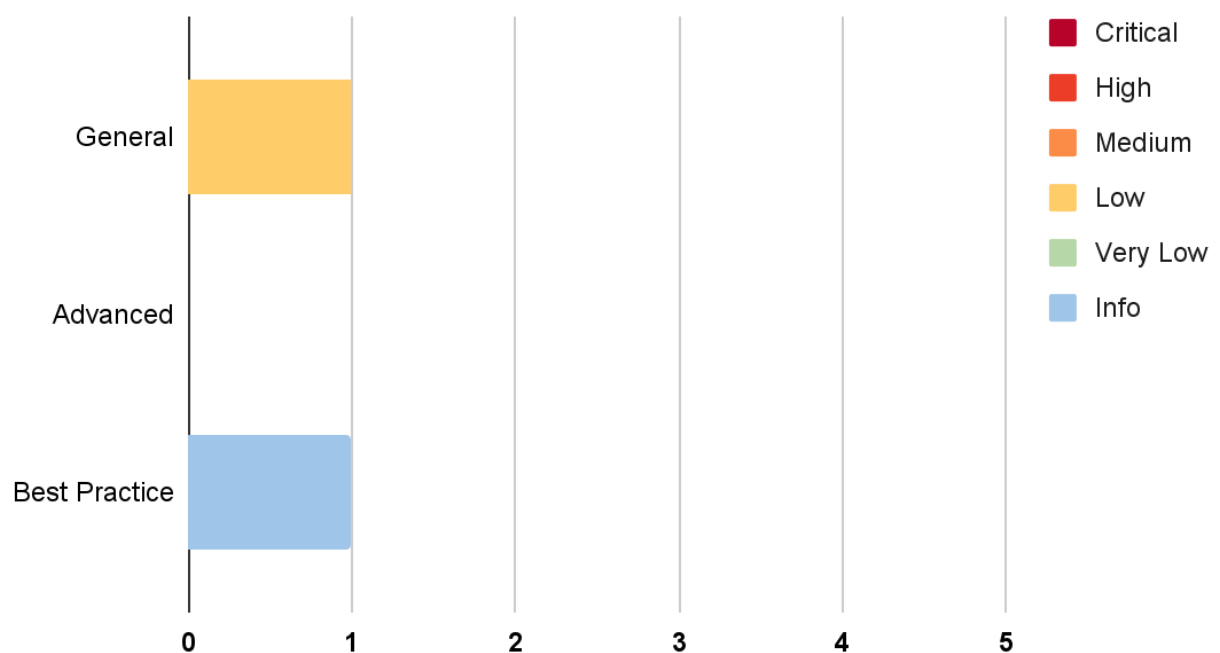
## 4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

### Assessment:



### Reassessment:



The statuses of the issues are defined as follows:

Status	Description
<b>Resolved</b>	The issue has been resolved and has no further complications.
<b>Resolved *</b>	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
<b>Acknowledged</b>	The issue's risk has been acknowledged and accepted.
<b>No Security Impact</b>	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Transaction Ordering Dependence	General	<b>High</b>	<b>Resolved *</b>
IDX-002	Centralized Control of State Variable	General	<b>High</b>	<b>Resolved *</b>
IDX-003	Unchecked Return Value of ERC20 transferFrom()	General	<b>Medium</b>	<b>Resolved</b>
IDX-004	Unbound Configuration Parameter	Advanced	<b>Low</b>	<b>Resolved</b>
IDX-005	Incorrect Order of Operations	Advanced	<b>Low</b>	<b>Resolved</b>
IDX-006	Smart Contract with Unpublished Source Code	General	<b>Low</b>	<b>Acknowledged</b>
IDX-007	Insufficient Logging for Privileged Functions	General	<b>Very Low</b>	<b>Resolved</b>
IDX-008	Improper Function Visibility	Best Practice	<b>Info</b>	<b>No Security Impact</b>
IDX-009	Use of Duplicate Literals	Best Practice	<b>Info</b>	<b>Resolved</b>
IDX-010	Inexplicit Solidity Compiler Version	Best Practice	<b>Info</b>	<b>Resolved</b>
IDX-011	Unnecessary Function due to Duplicate Functionality	Best Practice	<b>Info</b>	<b>Resolved</b>

\* The mitigations or clarifications by DAgora can be found in Chapter 5.

## 5. Detailed Findings Information

### 5.1. Transaction Ordering Dependence

ID	IDX-001
Target	DagoraMarketplace
Category	General Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	<p><b>Severity: High</b></p> <p><b>Impact: High</b> The registered owner of <b>RoyaltyFee</b> collection can instantly raise the royalty fee during the purchase from the buyers, resulting in a lower received amount for the seller of the affected NFT collection.</p> <p><b>Likelihood: Medium</b> Only the owner of the <b>RoyaltyFee</b> collection and an admin of the <b>DagoraMarketplace</b> contract can set the royalty fee of each collection.</p>
Status	<p><b>Resolved *</b></p> <p>The DAGora team has mitigated the issue by implementing a boundary of the possible royalty fee value to reduce the impact of this issue.</p>

#### 5.1.1. Description

The **DagoraMarketplace** contract is an NFT marketplace that allows anyone to buy and sell the NFTs. For the NFT creator, the **DagoraMarketplace** contract allows the NFT creators to register their NFTs to the platform through the **setRoyaltyFeeOwner()** function in order to collect the royalty fee, a fee that occurs for any successful buy and sell transactions. The set fee value can be set up to 99.99 percent.

#### DagoraMarketplace.sol

```

1350 function setRoyaltyFeeOwner(address _collection, bytes32 _byteCodeHash, uint256
    _nonce, bool _isCreate2, uint32 _fee, uint256 _expiresAt) external override {
1351     require(_fee < 10000, "Dagora Marketplace: Invalid input");
1352     uint256 royaltyFeeId = uint256(uint160(_collection));
1353     address owner = _getOwnerOfToken(dagoraRoyaltyFee, royaltyFeeId);
1354     if (owner == address(0)) {
1355         require(_collection == _getContractAddress(_byteCodeHash, _nonce,
    _isCreate2), "Dagora Marketplace: Not creator of collection");
1356         IDagora(dagoraRoyaltyFee).mint(msg.sender, royaltyFeeId);
1357     } else {
1358         require(_getOwnerOfToken(dagoraRoyaltyFee, royaltyFeeId) == msg.sender,

```

```

1359     "DAgora Marketplace: Not owner of royalty fee");
1360 }
1361     _royaltyFeeConfigs[royaltyFeeId].fee = _fee;
1362     _royaltyFeeConfigs[royaltyFeeId].expiresAt = _expiresAt;
1363     emit UpdateRoyaltyFee(_collection, _fee, _expiresAt);
1364 }

```

When a user buys an NFT collection which is already registered as the **RoyaltyFee** collection through the **buy()** function, a part of the NFT's sold amount will be deducted and sent to the owner of the **RoyaltyFee** collection at line 1470 through the **\_subRoyaltyFee()** function.

### DagoraMarketplace.sol

```

1444 function buy(address[] calldata _metaAddress, uint256[] calldata _metaUInt,
bytes memory _signature) public payable isUnuseSignature(_signature)
isInitPackage(_metaAddress[0]) isValidTime(_metaUInt[1] + _metaUInt[2]) {
1445     require(_verifySignature(_metaAddress, _metaUInt, _signature,
_metaAddress[1]), "DAgora Marketplace: Signature not match");
1446     require(_metaAddress.length >= 4, "DAgora Marketplace: Invalid Input");
1447     require(_metaAddress.length + 1 == _metaUInt.length, "DAgora Marketplace:
Invalid Input");
1448
1449     address[] memory tokenSaleList = new address[](_metaAddress.length - 3);
1450     uint256[] memory tokenIdList = new uint256[](_metaUInt.length - 4);
1451
1452     for (uint256 i = 0; i < _metaAddress.length - 3; i++) {
1453         tokenSaleList[i] = _metaAddress[i + 3];
1454         tokenIdList[i] = _metaUInt[i + 4];
1455     }
1456
1457     if (_metaAddress[2] != address(0)) {
1458         require(_metaAddress[2] == msg.sender, "DAgora Marketplace: Only
reserve address can make this payment");
1459     }
1460
1461     // get amount to contract
1462     if (_metaAddress[0] == address(0)) {
1463         require(msg.value >= _metaUInt[0], "DAgora Marketplace: Not enough
payment");
1464     } else {
1465         IERC20(_metaAddress[0]).safeTransferFrom(msg.sender, address(this),
_metaUInt[0]);
1466     }
1467
1468     // transfer Royalty fee: buyer -> creator
1469     uint256 amountAfterSubMarketFee = _subMarketFee(_metaUInt[0],

```

```

_packageInfos[_metaAddress[0]].claimFee,
_packageInfos[_metaAddress[0]].marketFee);
1470     uint256 amountAfterSubRoyaltyFee = _subRoyaltyFee(tokenSaleList,
_metaAddress[0], amountAfterSubMarketFee);
1471
1472     // transfer NFT: seller -> buyer
1473     _safeTransferNFT(tokenSaleList, tokenIdList, _metaAddress[1], msg.sender);
1474
1475     // transfer token: contract -> seller
1476     if (_metaAddress[0] == address(0)) {
1477         payable(_metaAddress[1]).transfer(amountAfterSubRoyaltyFee);
1478     } else {
1479         IERC20(_metaAddress[0]).safeTransfer(_metaAddress[1],
amountAfterSubRoyaltyFee);
1480     }
1481
1482     _disableSignature(_signature);
1483
1484     emit Buy(msg.sender, _metaAddress, _metaUInt);
1485 }

```

The deducted amount is based on the current royalty fee configuration as in lines 1280 and 1290.

### DagoraMarketplace.sol

```

1273 function _subRoyaltyFee(address[] memory _collections, address _tokenAddress,
uint _amount) private returns(uint256) {
1274     uint256 subAmount = _amount;
1275     uint256 totalRoyaltyFee = 0;
1276
1277     if (_isSameCollection(_collections)) {
1278         uint256 royaltyFeeId = uint256(uint160(_collections[0]));
1279         if (_isValidRoyaltyFee(royaltyFeeId)) {
1280             totalRoyaltyFee =
subAmount.mul(_royaltyFeeConfigs[royaltyFeeId].fee).div(PERCENT);
1281             _transferRoyaltyFee(_collections[0], _tokenAddress,
totalRoyaltyFee, _royaltyFeeConfigs[royaltyFeeId].fee);
1282         }
1283     } else {
1284         totalRoyaltyFee =
subAmount.mul(_packageInfos[_tokenAddress].totalRoyaltyFee).div(PERCENT);
1285         uint256 totalPercent = 0;
1286
1287         for (uint i = 0; i < _collections.length; i++) {
1288             uint256 royaltyFeeId = uint256(uint160(_collections[i]));
1289             if (_isValidRoyaltyFee(royaltyFeeId)) {
1290                 totalPercent =
totalPercent.add(_royaltyFeeConfigs[royaltyFeeId].fee);

```

```
1291     }
1292 }
1293
1294 if (totalPercent == 0) {
1295     return _amount;
1296 }
1297
1298 for (uint i = 0; i < _collections.length; i++) {
1299     if (!_isTransferRoyaltyFee[_collections[i]]) {
1300         _transferRoyaltyFee(_collections[i], _tokenAddress,
totalRoyaltyFee, totalPercent);
1301     }
1302     _isTransferRoyaltyFee[_collections[i]] = true;
1303 }
1304 _clearTempTransferRoyaltyFee(_collections);
1305 }
1306
1307
1308 return _amount.sub(totalRoyaltyFee);
1309 }
```

Hence, the owner of the **RoyaltyFee** collection can raise the royalty fee before any buying transaction is successful. This results in the lower sold amount for the seller which is taken by the owner of the **RoyaltyFee** collection.

### 5.1.2. Remediation

Inspex suggests applying the active royalty fee configuration to the signed signature from the seller during the NFT listing, so if the royalty fee is changed, the signed signature will be invalid, resulting in the seller having to sign the signature again in order to sell it (accept the new royalty fee).

However, with this solution, there will be several changes to the source code, so one of the following options, or both, can be used to mitigate this issue:

- Reducing the cap of royalty fee configuration to the reasonable amount such as 10% as maximum. This will reduce the impact of this issue.
- Allowing only reducing the royalty fee while it is active. However, the owner of **RoyaltyFee** collection can still front-run attack the first buy transaction when the previous royalty fee configuration is expired by renewing the royalty fee configuration again.

## 5.2. Centralized Control of State Variable

ID	IDX-002
Target	DagoraMarketplace
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	<b>Severity: High</b> <b>Impact: High</b> The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. <b>Likelihood: Medium</b> There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	<b>Resolved *</b> The DAGora team has mitigated this issue by applying the mapping to track the listed NFTs for auction in the marketplace, so the <code>withdrawNFT()</code> function will only work to withdraw NFTs for users who transfer NFTs to this address mistakenly. For the rest of the mentioned functions, there is a cap to limit their impact, which is negligible. This is mitigated.

### 5.2.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

Target	Function	Modifier
DagoraMarketplace.sol (L:963)	addAdmin()	onlyOwner
DagoraMarketplace.sol (L:967)	removeAdmin()	onlyOwner
DagoraMarketplace.sol (L:1337)	setDagoraRoyaltyFee()	onlyOwner
DagoraMarketplace.sol (L:1374)	setRoyaltyFeeAdmin()	onlyAdmin
DagoraMarketplace.sol (L:1389)	changePaymentToken	onlyOwner

DagoraMarketplace.sol (L:1394)	registerPackage()	onlyOwner
DagoraMarketplace.sol (L:1404)	configurePackage()	onlyOwner
DagoraMarketplace.sol (L:1413)	unRegisterPackage()	onlyOwner
DagoraMarketplace.sol (L:1424)	configureFixedVariable()	onlyOwner
DagoraMarketplace.sol (L:1632)	withdrawNFT()	onlyOwner
DagoraMarketplace.sol (L:1641)	withdraw()	onlyOwner

### 5.2.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run smart contract governance to control the use of these functions
- Using a timelock mechanism to delay the changes for a reasonable amount of time, e.g., 24 hours

Please note that, if the timelock mechanism is applied, since the **endBid()** function will be regularly used by the admin in order to close the auction, it is suggested to change the allowed caller of this function to another instead of **onlyAdmin** modifier to prevent it from being delayed.

### 5.3. Unchecked Return Value of ERC20 transferFrom()

ID	IDX-003
Target	DagoraMarketplace
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	<p><b>Severity: Medium</b></p> <p><b>Impact: High</b> When the <code>transferFrom()</code> function fails, it returns <code>false</code> rather than reverting, allowing an attacker to execute the <code>listing()</code> function without paying a fee.</p> <p><b>Likelihood: Low</b> This issue is unlikely to occur since it requires the platform to add the non-standard ERC20 token as the accepted listing fee token. The reason is that most of the commonly used tokens for the marketplace to add as the listing fee are implemented according to the ERC20 standard, which reverts a transaction in the event of a failure.</p>
Status	<p><b>Resolved</b></p> <p>The DAGora team has resolved this issue by changing the use of <code>transferFrom()</code> function into <code>safeTransferFrom()</code> function for transferring the ERC20 token.</p>

#### 5.3.1. Description

ERC20 tokens can be implemented in multiple ways, for example, allowing the execution of failed `transferFrom()` functions by returning `false` instead of reverting when the invalid transfer amount occurs.

In the `DagoraMarketplace` contract, the `listing()` function accepts the NFT with the payment token. The payment token will be transferred to the platform as the listing fee in line 1544.

#### DagoraMarketplace.sol

```

1523 function listing(address _saleByToken, address[] calldata _tokenAddresses,
1524   uint256[] calldata _tokenIds, uint256 _startPrice, uint256 _endPrice, uint256
   _expiresAt) public isValidTime(_expiresAt) {
1525     require (_tokenAddresses.length == _tokenIds.length, "Dagora Marketplace:
   Invalid Input");
1526     require(_saleByToken != address(0), "Dagora Marketplace: Native token not
   support");
1527     bytes32 nftHash = keccak256(abi.encodePacked(_tokenAddresses, _tokenIds));
1528
1529     require(_listingInfos[nftHash].seller == address(0), "Dagora Marketplace:
   List item already listed on dagora");

```

```
1530
1531     _listingInfos[nftHash] = ListingInfo({
1532         seller: msg.sender,
1533         saleByToken: _saleByToken,
1534         startPrice: _startPrice,
1535         endPrice: _endPrice,
1536         expiresAt: _expiresAt
1537     });
1538
1539     PackageInfo memory package = _packageInfos[_saleByToken];
1540
1541     // get listing fee
1542     if (package.listingFee > 0) {
1543         IERC20 token = IERC20(_saleByToken);
1544         token.transferFrom(msg.sender, address(this), package.listingFee);
1545     }
1546
1547     //transfer nft to marketplace
1548     _safeTransferNFT(_tokenAddresses, _tokenIds, msg.sender, address(this));
1549
1550     emit ListingNFT(msg.sender, _tokenAddresses, _tokenIds, _startPrice,
1551 _expiresAt);
1552 }
```

The return value of the `transferFrom()` function is not checked, so the transfer transaction of tokens that return false on failure will not be reverted.

### 5.3.2. Remediation

Inspex suggests replacing the `transferFrom()` function with the `safeTransferFrom()` function from OpenZeppelin's `SafeERC20` library, for example:

#### DagoraMarketplace.sol

```
1523 function listing(address _saleByToken, address[] calldata _tokenAddresses,
1524 uint256[] calldata _tokenIds, uint256 _startPrice, uint256 _endPrice, uint256
1525 _expiresAt) public isValidTime(_expiresAt) {
1526     require (_tokenAddresses.length == _tokenIds.length, "Dagora Marketplace:
1527 Invalid Input");
1528     require(_saleByToken != address(0), "Dagora Marketplace: Native token not
1529 support");
1530     bytes32 nftHash = keccak256(abi.encodePacked(_tokenAddresses, _tokenIds));
1531
1532     require(_listingInfos[nftHash].seller == address(0), "Dagora Marketplace:
1533 List item already listed on dagora");
1534 }
```

```
1531     _listingInfos[nftHash] = ListingInfo({
1532         seller: msg.sender,
1533         saleByToken: _saleByToken,
1534         startPrice: _startPrice,
1535         endPrice: _endPrice,
1536         expiresAt: _expiresAt
1537     });
1538
1539     PackageInfo memory package = _packageInfos[_saleByToken];
1540
1541     // get listing fee
1542     if (package.listingFee > 0) {
1543         IERC20 token = IERC20(_saleByToken);
1544         token.safeTransferFrom(msg.sender, address(this), package.listingFee);
1545     }
1546
1547     // transfer nft to marketplace
1548     _safeTransferNFT(_tokenAddresses, _tokenIds, msg.sender, address(this));
1549
1550     emit ListingNFT(msg.sender, _tokenAddresses, _tokenIds, _startPrice,
1551 _expiresAt);
1551 }
```

## 5.4. Unbound Configuration Parameter

ID	IDX-004
Target	DagoraMarketplace
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<p><b>Severity:</b> Low</p> <p><b>Impact:</b> Medium The registered packages can have their <b>marketFee</b> attribute nearly equal to or exceed the <b>PERCENT</b> constant, which can cause the token amount of the selling NFTs to be wholly paid to the market fee.</p> <p><b>Likelihood:</b> Low Only the owner of the <b>DagoraMarketplace</b> contract can set the <b>marketFee</b> attribute of each package (payment token). The configuration of the fee affects every sale within the same package. It cannot target a specific sale.</p>
Status	<p><b>Resolved</b> The DAGora team has resolved this issue by implementing a boundary for the <b>marketFee</b> and <b>totalRoyaltyFee</b> values. For the boundary of the values, the DAGora team has configured both values at 20 percent.</p>

### 5.4.1. Description

The **DagoraMarketplace** contract offers two ways to exchange the NFT assets: Selling and Listing. When the exchange happens, basically, the buyer (or the final bidder, in the listing case) will transfer the agreed amount to the seller. The platform will subtract the market fee from the buyer's paid amount, and then the leftover amount will have the royalty fee subtracted and sent to the owner of the **RoyaltyFee** collection. Finally, the remaining amount from subtracting fees will be sent to the seller.

The owner can increase the market fee up to 100 percent of the buyer paid amount to take all of the amounts into the market fee and leave nothing to the creator and the seller.

#### DagoraMarketplace.sol

```

1131 function _subMarketFee(uint256 _amount, uint256 _claimFee, uint256 _marketFee)
    private pure returns(uint256) {
1132     uint256 amount = _amount;
1133     if (_claimFee > 0) {
1134         amount = amount.sub(_claimFee);
1135     }
1136     if (_marketFee > 0) {

```

```
1137         amount = amount.sub(amount.div(PERCENT).mul(_marketFee));
1138     }
1139
1140     return amount;
1141 }
```

The `registerPackage()` and `configurePackage()` function are the only functions that can set the market fee related variables. These functions do not have any restrictions to limit the amount of those fees. The `packageInfo.marketFee` value is the percentage fee that the platform will take from the sellers. The `packageInfo.claimFee` value is also the fee that the platform will take from the sellers similar to the `packageInfo.marketFee` value, but it is a flat fee instead of a percentage fee. So, if the selling amount is less than the `packageInfo.claimFee` value, the buying transaction will be reverted.

#### DagoraMarketplace.sol

```
1393 // @dev Register Package for token can sell in dagora
1394 function registerPackage(address _token, uint256 _marketFee, uint256 _claimFee,
1395     uint256 _totalRoyaltyFee) public onlyOwner() {
1396     PackageInfo storage packageInfo = _packageInfos[_token];
1397
1398     packageInfo.isInitial = true;
1399     packageInfo.marketFee = _marketFee;
1400     packageInfo.claimFee = _claimFee;
1401     packageInfo.totalRoyaltyFee = _totalRoyaltyFee;
1402 }
1403 // @dev Config Package for token can sell in dagora
1404 function configurePackage(address _token, uint256 _marketFee, uint256
1405     _claimFee, uint256 _totalRoyaltyFee) public isInitPackage(_token) onlyOwner() {
1406     PackageInfo storage packageInfo = _packageInfos[_token];
1407
1408     packageInfo.marketFee = _marketFee;
1409     packageInfo.claimFee = _claimFee;
1410     packageInfo.totalRoyaltyFee = _totalRoyaltyFee;
1411 }
```

For the `packageInfo.totalRoyaltyFee` value, it is the total royalty fee in the case of selling a multiple collection NFT. If the sum of the `packageInfo.totalRoyaltyFee` value and the `packageInfo.marketFee` value is 100 percent, the seller will get nothing from the selling.

#### 5.4.2. Remediation

Inspex suggests enforcing a boundary value of these states when they are set by declaring constant states for the highest possible value of each state. The appropriate values of each state should be determined by the business model of DAGora.

For example, setting the boundary value of the `marketFee` and the `totalRoyaltyFee` states to 20 percent. For the `claimFee` value, according to the business design, it is suggested to set the `claimFee` value as low as possible such as 10 tokens since the buying of NFT with a price lower than the `claimFee` value will fail.

### DagoraMarketplace.sol

```
1393 uint256 constant MARKET_FEE_CAP = 2000;
1394 uint256 constant CLAIM_FEE_CAP = 10;
1395 uint256 constant TOTAL_ROYALTY_FEE_CAP = 2000;
1396
1397
1398 // @dev Register Package for token can sell in dagora
1399 function registerPackage(address _token, uint256 _marketFee, uint256 _claimFee,
uint256 _totalRoyaltyFee) public onlyOwner() {
1400     require(_marketFee <= MARKET_FEE_CAP, "Dagora Marketplace: Invalid market
fee");
1401     require(_claimFee <= CLAIM_FEE_CAP, "Dagora Marketplace: Invalid claim
fee");
1402     require(_totalRoyaltyFee <= TOTAL_ROYALTY_FEE_CAP, "Dagora Marketplace:
Invalid total royalty fee");
1403
1404     PackageInfo storage packageInfo = _packageInfos[_token];
1405
1406     packageInfo.isInitial = true;
1407     packageInfo.marketFee = _marketFee;
1408     packageInfo.claimFee = _claimFee;
1409     packageInfo.totalRoyaltyFee = _totalRoyaltyFee;
1410 }
1411
1412 // @dev Config Package for token can sell in dagora
1413 function configurePackage(address _token, uint256 _marketFee, uint256
_claimFee, uint256 _totalRoyaltyFee) public isInitPackage(_token) onlyOwner() {
1414     require(_marketFee <= MARKET_FEE_CAP, "Dagora Marketplace: Invalid market
fee");
1415     require(_claimFee <= CLAIM_FEE_CAP, "Dagora Marketplace: Invalid claim
fee");
1416     require(_totalRoyaltyFee <= TOTAL_ROYALTY_FEE_CAP, "Dagora Marketplace:
Invalid total royalty fee");
1417
1418     PackageInfo storage packageInfo = _packageInfos[_token];
1419
1420     packageInfo.marketFee = _marketFee;
1421     packageInfo.claimFee = _claimFee;
1422     packageInfo.totalRoyaltyFee = _totalRoyaltyFee;
1423 }
```

## 5.5. Incorrect Order of Operations

ID	IDX-005
Target	DagoraMarketplace
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<b>Severity: Low</b> <b>Impact: Low</b> The rounding error from the division before multiplication can cause the fees to be slightly miscalculated. <b>Likelihood: Medium</b> Since the price of NFT can be set to any number, it is likely that the fee calculation will result in a floating number that is rounded off from the division before applying the multiplication.
Status	<b>Resolved</b> The DAgora team has resolved this issue by reordering the operation order in the <code>_subMarketFee()</code> and the <code>_transferRoyaltyFee()</code> functions.

### 5.5.1. Description

Solidity supports only integer values but not floating point values. The division of integers can result in a value with decimal points, which will be rounded off. This rounding error can cause the calculation to be different from what it should be, especially when that value is later multiplied with another value.

In the `_subMarketFee()` function, the `amount` variable is calculated by subtracting the `amount` with the value by dividing the amount with `PERCENT`, before multiplying it by `_marketFee`. The rounding error caused by the division is amplified in the multiplication and can increase the amount of miscalculation.

#### DagoraMarketplace.sol

```
1131 function _subMarketFee(uint256 _amount, uint256 _claimFee, uint256 _marketFee)
    private pure returns(uint256) {
1132     uint256 amount = _amount;
1133     if (_claimFee > 0) {
1134         amount = amount.sub(_claimFee);
1135     }
1136     if (_marketFee > 0) {
1137         amount = amount.sub(amount.div(PERCENT).mul(_marketFee));
1138     }
1139
1140     return amount;
```

```
1141 }
```

In the `_transferRoyaltyFee()` function, the `payment` variable is calculated from multiplying the `feePercent` in line 1255. The `feePercent` result comes from the division of `_totalPercent` value, so the miscalculation will occur due to the rounded off division result being amplified by the multiplication.

#### DagoraMarketplace.sol

```
1248 function _transferRoyaltyFee(address _collection, address _tokenAddress,
1249     uint256 _amount, uint256 _totalPercent) private {
1250     uint256 royaltyFeeId = uint256(uint160(_collection));
1251     if (_isValidRoyaltyFee(royaltyFeeId)) {
1252         uint256 fee = _royaltyFeeConfigs[royaltyFeeId].fee;
1253
1254         uint256 feePercent = fee.mul(PERCENT).div(_totalPercent);
1255         uint256 payment = _amount.mul(feePercent).div(PERCENT);
1256         address ownerAddress = _getOwnerOfToken(dagoraRoyaltyFee,
royaltyFeeId);
1257         if (_tokenAddress != address(0)) {
1258             IERC20 token = IERC20(_tokenAddress);
1259             token.safeTransfer(ownerAddress, payment);
1260         } else {
1261             payable(ownerAddress).transfer(payment);
1262         }
1263     }
1264 }
```

The lines of code with division before multiplication are as follows:

- DagoraMarketplace.sol (L:1137)
- DagoraMarketplace.sol (L:1254 - 1255)

### 5.5.2. Remediation

Inspex suggests modifying the affected lines of code to perform multiplication before division, for example:

#### DagoraMarketplace.sol

```
1131 function _subMarketFee(uint256 _amount, uint256 _claimFee, uint256 _marketFee)
1132     private pure returns(uint256) {
1133     uint256 amount = _amount;
1134     if (_claimFee > 0) {
1135         amount = amount.sub(_claimFee);
1136     }
1137     if (_marketFee > 0) {
1138         amount = amount.sub(amount.mul(_marketFee).div(PERCENT));
1139     }
1140 }
```

```
1139
1140     return amount;
1141 }
```

#### DagoraMarketplace.sol

```
1248 function _transferRoyaltyFee(address _collection, address _tokenAddress,
uint256 _amount, uint256 _totalPercent) private {
1249     uint256 royaltyFeeId = uint256(uint160(_collection));
1250
1251     if (_isValidRoyaltyFee(royaltyFeeId)) {
1252         uint256 fee = _royaltyFeeConfigs[royaltyFeeId].fee;
1253
1254         uint256 payment = _amount.mul(fee).div(_totalPercent);
1255         address ownerAddress = _getOwnerOfToken(dagoraRoyaltyFee,
royaltyFeeId);
1256         if (_tokenAddress != address(0)) {
1257             IERC20 token = IERC20(_tokenAddress);
1258             token.safeTransfer(ownerAddress, payment);
1259         } else {
1260             payable(ownerAddress).transfer(payment);
1261         }
1262     }
1263 }
```

## 5.6. Smart Contract with Unpublished Source Code

ID	IDX-006
Target	DAgoraCollection DagoraMarketplace
Category	General Smart Contract Vulnerability
CWE	CWE-1006: Bad Coding Practices
Risk	<b>Severity: Low</b>  <b>Impact: Medium</b> The logic of the smart contract may not align with the user's understanding, causing undesired actions to be taken when the user interacts with the smart contract.  <b>Likelihood: Low</b> The possibility for the users to misunderstand the functionalities of the contract is not very high with the help of the documentation and user interface.
Status	<b>Acknowledged</b> The Coin98 team has acknowledged this issue and decided not to publish the source code because the team wants to protect their intellectual property.

### 5.6.1. Description

The smart contract source code is not publicly published, so the users will not be able to easily verify the correctness of the functionalities and the logic of the smart contract by themselves. Therefore, it is possible that the user's understanding of the smart contract does not align with the actual implementation, leading to undesired actions on interacting with the smart contract.

### 5.6.2. Remediation

Inspex suggests publishing the contract source code through a public code repository or verifying the smart contract source code on the blockchain explorer so that the users can easily read and understand the logic of the smart contract by themselves.

## 5.7. Insufficient Logging for Privileged Functions

ID	IDX-007
Target	DAGoraCollection DagoraMarketplace
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	<b>Severity:</b> <b>Very Low</b>  <b>Impact:</b> <b>Low</b> Privileged functions' executions cannot be monitored easily by the users.  <b>Likelihood:</b> <b>Low</b> It is not likely that the execution of the privileged functions will be a malicious action.
Status	<b>Resolved</b> The DAGora team has resolved this issue by emitting events to the list of functions.

### 5.7.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

For example, the owner can change the `dagoraRoyaltyFee` state of the contract by executing the `setDagoraRoyaltyFee()` function in the `DagoraMarketplace` contract, and no events are emitted.

#### DagoraMarketplace.sol

```
1337 function setDagoraRoyaltyFee(address _dagoraRoyaltyFee) public onlyOwner {
1338     dagoraRoyaltyFee = _dagoraRoyaltyFee;
1339 }
```

The privileged functions without sufficient logging are as follows:

File	Contract	Function
DagoraCollection.sol (L:1499)	DAGoraCollection	setBaseURI()
DagoraMarketplace.sol (L:963)	DagoraMarketplace	addAdmin()
DagoraMarketplace.sol (L:967)	DagoraMarketplace	removeAdmin()
DagoraMarketplace.sol (L:1337)	DagoraMarketplace	setDagoraRoyaltyFee()

DagoraMarketplace.sol (L:1389)	DagoraMarketplace	changePaymentToken()
DagoraMarketplace.sol (L:1394)	DagoraMarketplace	registerPackage()
DagoraMarketplace.sol (L:1404)	DagoraMarketplace	configurePackage()
DagoraMarketplace.sol (L:1413)	DagoraMarketplace	unRegisterPackage()
DagoraMarketplace.sol (L:1424)	DagoraMarketplace	configureFixedVariable()
DagoraMarketplace.sol (L:1505)	DagoraMarketplace	updateListingInfo()

### 5.7.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

#### DagoraMarketplace.sol

```
1337 event SetDagoraRoyaltyFee(address _dagoraRoyaltyFee);
1338 function setDagoraRoyaltyFee(address _dagoraRoyaltyFee) public onlyOwner {
1339     dagoraRoyaltyFee = _dagoraRoyaltyFee;
1340     emit SetDagoraRoyaltyFee(_dagoraRoyaltyFee);
1341 }
```

## 5.8. Improper Function Visibility

ID	IDX-008
Target	DAGoraCollection DagoraMarketplace
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	<b>Severity:</b> Info <b>Impact:</b> None <b>Likelihood:</b> None
Status	<b>No Security Impact</b> The DAGora team has partially resolved the issue by changing the modifiers of some of the listed functions.

### 5.8.1. Description

Functions with public visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

The following source code shows that the `totalToken()` function of the `DAGoraCollection` is set to public and it is never called from any internal function.

#### DagoraCollection.sol

```
1503 function totalToken() public view returns (uint256) {  
1504     return _tokenIdTracker.current();  
1505 }
```

The following table contains all functions that have public visibility and are never called from any internal function.

Target	Contract	Function
DagoraCollection.sol (L:895)	DAGoraCollection	rawOwnerOf()
DagoraCollection.sol (L:902)	DAGoraCollection	name()
DagoraCollection.sol (L:909)	DAGoraCollection	symbol()
DagoraCollection.sol (L:916)	DAGoraCollection	tokenURI()
DagoraCollection.sol (L:935)	DAGoraCollection	approve()

DagoraCollection.sol (L:959)	DAgoraCollection	setApprovalForAll()
DagoraCollection.sol (L:976)	DAgoraCollection	transferFrom()
DagoraCollection.sol (L:990)	DAgoraCollection	safeTransferFrom()
DagoraCollection.sol (L:1281)	DAgoraCollection	tokenByIndex()
DagoraCollection.sol (L:1455)	DAgoraCollection	renounceOwnership()
DagoraCollection.sol (L:1464)	DAgoraCollection	transferOwnership()
DagoraCollection.sol (L:1503)	DAgoraCollection	totalToken()
DagoraMarketplace.sol (L:929)	DagoraMarketplace	renounceOwnership()
DagoraMarketplace.sol (L:938)	DagoraMarketplace	transferOwnership()
DagoraMarketplace.sol (L:963)	DagoraMarketplace	addAdmin()
DagoraMarketplace.sol (L:967)	DagoraMarketplace	removeAdmin()
DagoraMarketplace.sol (L:1337)	DagoraMarketplace	setDagoraRoyaltyFee()
DagoraMarketplace.sol (L:1374)	DagoraMarketplace	setRoyaltyFeeAdmin()
DagoraMarketplace.sol (L:1389)	DagoraMarketplace	changePaymentToken()
DagoraMarketplace.sol (L:1404)	DagoraMarketplace	configurePackage()
DagoraMarketplace.sol (L:1413)	DagoraMarketplace	unRegisterPackage()
DagoraMarketplace.sol (L:1424)	DagoraMarketplace	configureFixedVariable()
DagoraMarketplace.sol (L:1433)	DagoraMarketplace	pay()
DagoraMarketplace.sol (L:1444)	DagoraMarketplace	buy()
DagoraMarketplace.sol (L:1491)	DagoraMarketplace	cancel()
DagoraMarketplace.sol (L:1505)	DagoraMarketplace	updateListingInfo()
DagoraMarketplace.sol (L:1523)	DagoraMarketplace	listing()
DagoraMarketplace.sol (L:1556)	DagoraMarketplace	cancelListing()
DagoraMarketplace.sol (L:1575)	DagoraMarketplace	endBid()
DagoraMarketplace.sol (L:1608)	DagoraMarketplace	getListingInfo()
DagoraMarketplace.sol (L:1616)	DagoraMarketplace	getPackageInfo()

DagoraMarketplace.sol (L:1624)	DagoraMarketplace	getRoyaltyFeeConfig()
DagoraMarketplace.sol (L:1632)	DagoraMarketplace	withdrawNFT()
DagoraMarketplace.sol (L:1641)	DagoraMarketplace	withdraw()

### 5.8.2. Remediation

Inspex suggests changing all functions' visibility to external if they are not called from any internal function as shown in the following example:

#### DagoraCollection.sol

```
1503 function totalToken() external view returns (uint256) {  
1504     return _tokenIdTracker.current();  
1505 }
```

## 5.9. Use of Duplicate Literals

ID	IDX-009
Target	DagoraMarketplace
Category	Smart Contract Best Practice
CWE	CWE-1106: Insufficient Use of Symbolic Constants
Risk	<b>Severity:</b> Info <b>Impact:</b> None <b>Likelihood:</b> None
Status	<b>Resolved</b> The DAgora team has resolved this issue by defining a constant parameter to handle fee's limit in the <code>setRoyaltyFeeOwner()</code> and the <code>setRoyaltyFeeAdmin()</code> functions.

### 5.9.1. Description

The use of literal numbers in the calculations can reduce the readability of the contract. There are literal numbers in the `DagoraMarketplace` contract that refer to the same value, i.e., the reference to a one hundred percent value, `PERCENT`. These numbers can be defined under the same constant variable to make the functions more readable and improve maintainability.

#### DagoraMarketplace.sol

```

1350 function setRoyaltyFeeOwner(address _collection, bytes32 _byteCodeHash, uint256
    _nonce, bool _isCreate2, uint32 _fee, uint256 _expiresAt) external override {
1351     require(_fee < 10000, "DAgora Marketplace: Invalid input");
1352     uint256 royaltyFeeId = uint256(uint160(_collection));
1353     address owner = _getOwnerOfToken(dagoraRoyaltyFee, royaltyFeeId);
1354     if (owner == address(0)) {
1355         require(_collection == _getContractAddress(_byteCodeHash, _nonce,
    _isCreate2), "DAgora Marketplace: Not creator of collection");
1356         IDagora(dagoraRoyaltyFee).mint(msg.sender, royaltyFeeId);
1357     } else {
1358         require(_getOwnerOfToken(dagoraRoyaltyFee, royaltyFeeId) == msg.sender,
    "DAgora Marketplace: Not owner of royalty fee");
1359     }
1360
1361     _royaltyFeeConfigs[royaltyFeeId].fee = _fee;
1362     _royaltyFeeConfigs[royaltyFeeId].expiresAt = _expiresAt;
1363     emit UpdateRoyaltyFee(_collection, _fee, _expiresAt);
1364 }

```

## DagoraMarketplace.sol

```

1374 function setRoyaltyFeeAdmin(address _collection, address _owner, uint32 _fee,
1375   uint256 _expiresAt) public onlyAdmin {
1376     require(_fee < 10000, "Dagora Marketplace: Invalid input");
1377     uint256 royaltyFeeId = uint256(uint160(_collection));
1378     address owner = _getOwnerOfToken(dagoraRoyaltyFee, royaltyFeeId);
1379
1379     if (owner == address(0)) {
1380         IDagora(dagoraRoyaltyFee).mint(_owner, royaltyFeeId);
1381     }
1382
1383     _royaltyFeeConfigs[royaltyFeeId].fee = _fee;
1384     _royaltyFeeConfigs[royaltyFeeId].expiresAt = _expiresAt;
1385     emit UpdateRoyaltyFee(_collection, _fee, _expiresAt);
1386 }

```

## 5.9.2. Remediation

Inspex suggests using the defined constant, PERCENT, for the value of the fee's limit in the setRoyaltyFeeOwner() and the setRoyaltyFeeAdmin() functions.

## DagoraMarketplace.sol

```

1350 function setRoyaltyFeeOwner(address _collection, bytes32 _byteCodeHash, uint256
1351   _nonce, bool _isCreate2, uint32 _fee, uint256 _expiresAt) external override {
1352     require(_fee < PERCENT, "Dagora Marketplace: Invalid input");
1353     uint256 royaltyFeeId = uint256(uint160(_collection));
1354     address owner = _getOwnerOfToken(dagoraRoyaltyFee, royaltyFeeId);
1355     if (owner == address(0)) {
1356         require(_collection == _getContractAddress(_byteCodeHash, _nonce,
1357   _isCreate2), "Dagora Marketplace: Not creator of collection");
1358         IDagora(dagoraRoyaltyFee).mint(msg.sender, royaltyFeeId);
1359     } else {
1360         require(_getOwnerOfToken(dagoraRoyaltyFee, royaltyFeeId) == msg.sender,
1361   "Dagora Marketplace: Not owner of royalty fee");
1362     }
1363
1364     _royaltyFeeConfigs[royaltyFeeId].fee = _fee;
1365     _royaltyFeeConfigs[royaltyFeeId].expiresAt = _expiresAt;
1366     emit UpdateRoyaltyFee(_collection, _fee, _expiresAt);
1367 }

```

## DagoraMarketplace.sol

```
1374 function setRoyaltyFeeAdmin(address _collection, address _owner, uint32 _fee,
1375 uint256 _expiresAt) public onlyAdmin {
1376     require(_fee < PERCENT, "Dagora Marketplace: Invalid input");
1377     uint256 royaltyFeeId = uint256(uint160(_collection));
1378     address owner = _getOwnerOfToken(dagoraRoyaltyFee, royaltyFeeId);
1379
1380     if (owner == address(0)) {
1381         IDagora(dagoraRoyaltyFee).mint(_owner, royaltyFeeId);
1382     }
1383
1384     _royaltyFeeConfigs[royaltyFeeId].fee = _fee;
1385     _royaltyFeeConfigs[royaltyFeeId].expiresAt = _expiresAt;
1386     emit UpdateRoyaltyFee(_collection, _fee, _expiresAt);
1387 }
```

## 5.10. Inexplicit Solidity Compiler Version

ID	IDX-010
Target	DAGoraCollection DagoraMarketplace
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	<b>Severity:</b> Info <b>Impact:</b> None <b>Likelihood:</b> None
Status	<b>Resolved</b> The DAGora team has resolved this issue by editing the Solidity compiler version to 0.7.6. The issue has been resolved.

### 5.10.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

Contract	Version
DAGoraCollection	^0.7.0
DagoraMarketplace	^0.7.0

#### DAGoraCollection.sol

```
1482 pragma solidity ^0.7.0;
```

### 5.10.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.7 is v0.7.6

(<https://github.com/ethereum/solidity/releases/tag/v0.7.6>).

#### DAGoraCollection.sol

```
1482 pragma solidity 0.7.6;
```

## 5.11. Unnecessary Function due to Duplicate Functionality

ID	IDX-011
Target	DagoraMarketplace
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	<b>Severity:</b> Info <b>Impact:</b> None <b>Likelihood:</b> None
Status	<b>Resolved</b> The DAGora team has resolved this issue by removing the <code>configurePackage()</code> function.

### 5.11.1. Description

At the current state, the `registerPackage()` function and the `configurePackage()` function have almost the same functionality. And the `registerPackage()` function can do exactly the same as the `configurePackage()` function can do but without a restriction from the `isInitPackage()` modifier. So, the `configurePackage()` function has no distinct purpose for use.

#### DagoraMarketplace.sol

```

1393 // @dev Register Package for token can sell in dagora
1394 function registerPackage(address _token, uint256 _marketFee, uint256 _claimFee,
1395   uint256 _totalRoyaltyFee) public onlyOwner() {
1396     PackageInfo storage packageInfo = _packageInfos[_token];
1397     packageInfo.isInitial = true;
1398     packageInfo.martketFee = _marketFee;
1399     packageInfo.claimFee = _claimFee;
1400     packageInfo.totalRoyaltyFee = _totalRoyaltyFee;
1401 }
1402
1403 // @dev Config Package for token can sell in dagora
1404 function configurePackage(address _token, uint256 _marketFee, uint256
1405   _claimFee, uint256 _totalRoyaltyFee) public isInitPackage(_token) onlyOwner() {
1406     PackageInfo storage packageInfo = _packageInfos[_token];
1407     packageInfo.martketFee = _marketFee;
1408     packageInfo.claimFee = _claimFee;
1409     packageInfo.totalRoyaltyFee = _totalRoyaltyFee;
1410 }

```

### 5.11.2. Remediation

Inspex suggests deleting the `configurePackage()` function. But, if these two functions really have a distinct purpose for use, we suggest adding event emitting for declaring the purposes of the functions and adding conditions to make these functions have different use cases. For example, add events for functions and add a condition in the `registerPackage()` function:

#### DagoraMarketplace.sol

```
1392 // @dev Register Package for token can sell in dagora
1393 event RegisterPackage(address _token, uint256 _marketFee, uint256 _claimFee,
1394   uint256 _totalRoyaltyFee);
1395 function registerPackage(address _token, uint256 _marketFee, uint256 _claimFee,
1396   uint256 _totalRoyaltyFee) public onlyOwner() {
1397     PackageInfo storage packageInfo = _packageInfos[_token];
1398     require(packageInfo.isInitial != true, "Dagora Marketplace: The package has
1399   already been set")
1400
1401     packageInfo.isInitial = true;
1402     packageInfo.martketFee = _marketFee;
1403     packageInfo.claimFee = _claimFee;
1404     packageInfo.totalRoyaltyFee = _totalRoyaltyFee;
1405
1406     emit RegisterPackage(_token, _marketFee, _claimFee, _totalRoyaltyFee);
1407 }
1408
1409 // @dev Config Package for token can sell in dagora
1410 event ConfigurePackage(address _token, uint256 _marketFee, uint256 _claimFee,
1411   uint256 _totalRoyaltyFee);
1412 function configurePackage(address _token, uint256 _marketFee, uint256
1413   _claimFee, uint256 _totalRoyaltyFee) public isInitPackage(_token) onlyOwner() {
1414     PackageInfo storage packageInfo = _packageInfos[_token];
1415
1416     packageInfo.martketFee = _marketFee;
1417     packageInfo.claimFee = _claimFee;
1418     packageInfo.totalRoyaltyFee = _totalRoyaltyFee;
1419
1420     emit ConfigurePackage(_token, _marketFee, _claimFee, _totalRoyaltyFee);
1421 }
```

## 6. Appendix

### 6.1. About Inspex



# CYBERSECURITY PROFESSIONAL SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

#### Follow Us On:

Website	<a href="https://inspex.co">https://inspex.co</a>
Twitter	<a href="https://twitter.com/InspexCo">@InspexCo</a>
Facebook	<a href="https://www.facebook.com/InspexCo">https://www.facebook.com/InspexCo</a>
Telegram	<a href="https://t.me/inspex_announcement">@inspex_announcement</a>



**inspex**  
CYBERSECURITY PROFESSIONAL SERVICE