Marketplace (EVM)

Smart Contract Audit Report
Prepared for DAgora

Date Issued:	Aug 21, 2023
Project ID:	AUDIT2022049
Version:	v1.0
Confidentiality Level:	Public

Report Information

Project ID	AUDIT2022049
Version	v1.0
Client	DAgora
Project	Marketplace (EVM)
Auditor(s)	Darunphop Pengkumta Fungkiat Phadejtaku
Author(s)	Darunphop Pengkumta Fungkiat Phadejtaku
Reviewer	Patipon Suwanbol
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Aug 21, 2023	Full report	Wachirawit Kanpanluk

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by DAgora, Inspex team conducted an audit to verify the security posture of the Marketplace (EVM) smart contracts between Nov 10, 2022 and Nov 11, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Marketplace (EVM) 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 1 critical, 1 high, 2 medium, 2 low, and 1 info-severity issues. With the project team's prompt response 1 critical, 1 high, 2 medium severity issues were resolved or mitigated in the reassessment, while 2 low and 1 info-severity issues were acknowledged by the team. Therefore, Inspex trusts that Marketplace (EVM) smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

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, 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 for the `DAgoraMarketplace` contract, and they will receive the `DAgoraCollection` NFTs as the owner representative.

Scope Information:

Project Name	Marketplace (EVM)
Website	https://dagora.xyz/
Smart Contract Type	Ethereum Smart Contract
Chain	Ethereum, BNB Smart Chain, Avalanche, Fantom, HECO Chain, Polygon, Arbitrum, and Boba Network
Programming Language	Solidity
Category	NFT, Marketplace

Audit Information:

Audit Method	Whitebox
Audit Date	Nov 10, 2022 - Nov 11, 2022
Reassessment Date	Nov 17, 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
DAgoraMarketplace	1a8b25f265c401b8c6f8fb06f84d8ce3adee45717dcc8be0b28120cf78954526

Reassessment

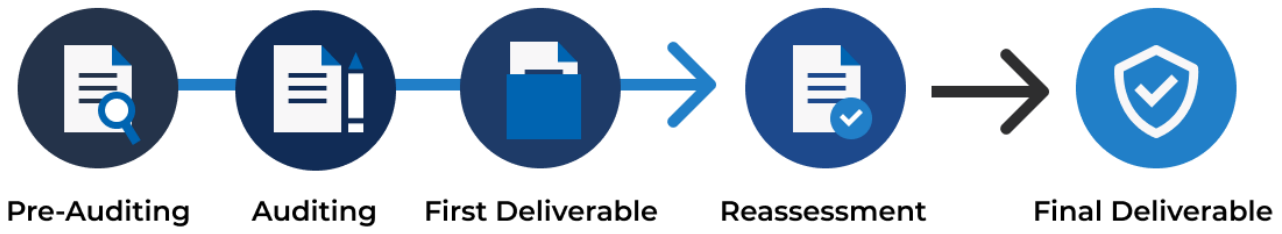
Contract	Bytecode SHA256 Hash
DAgoraMarketplace	2dc1ce931a3f0363823f9ed3fc41ae89552e93db4b098410dd29fb4c35855108

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 testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at https://inspex.gitbook.io/testing-guide/.

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

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**.

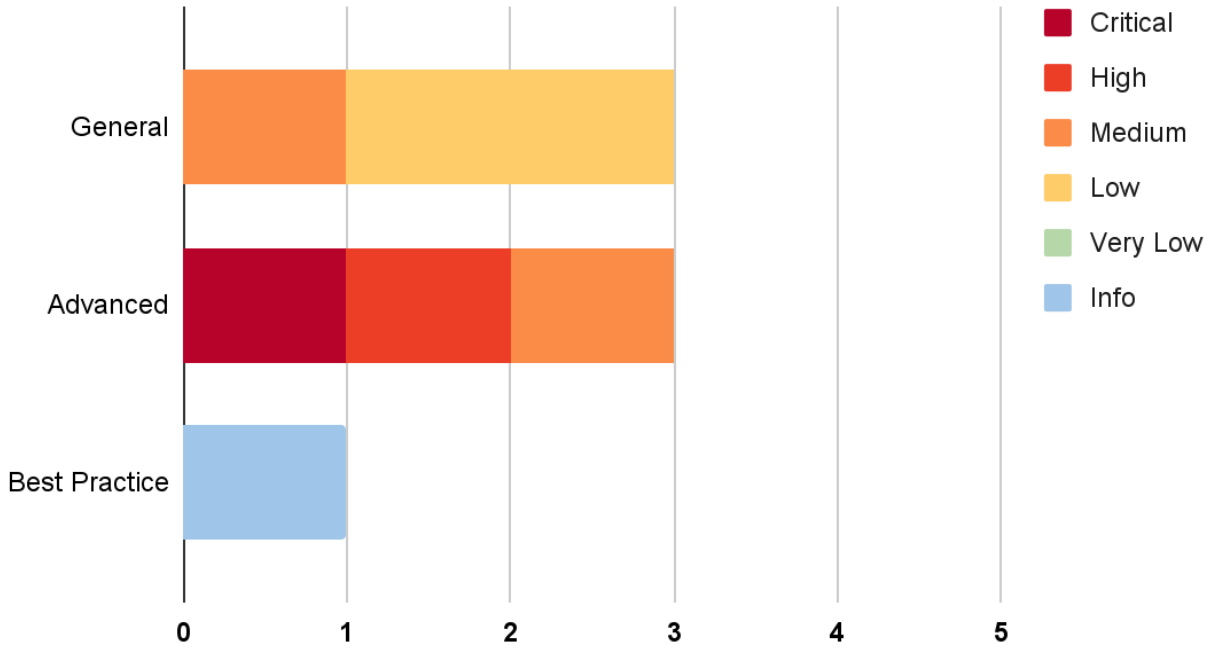
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would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

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

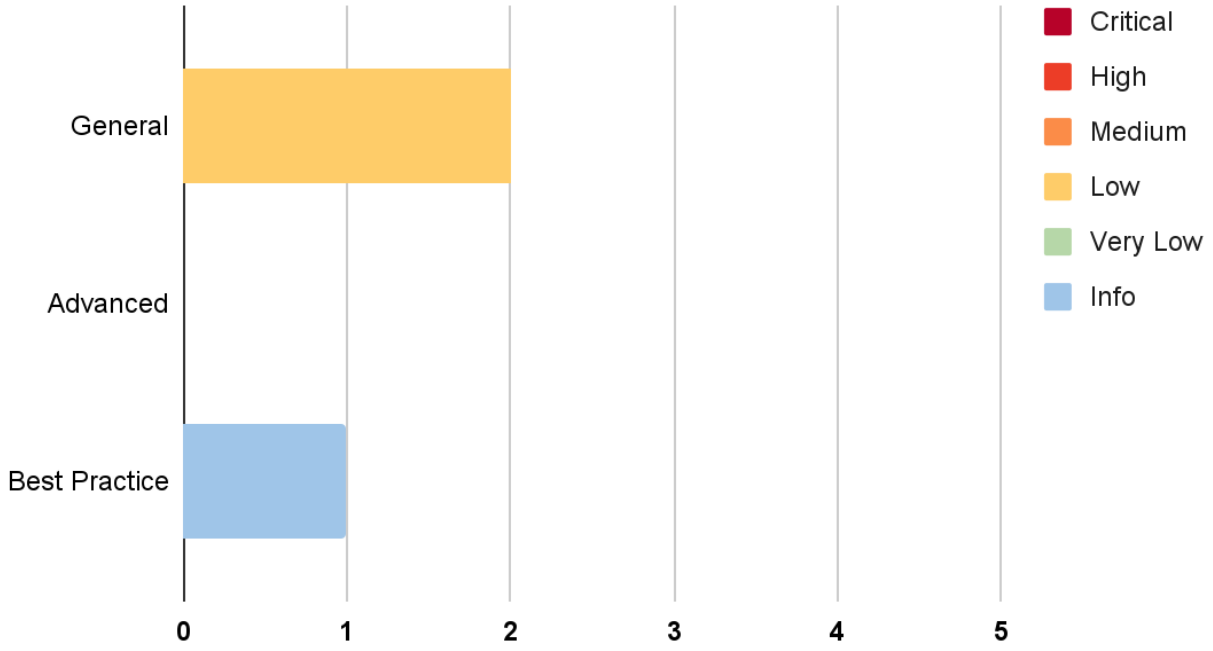
4. Summary of Findings

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
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	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	Invalid tokenSaleList Calculation in sell() Function	Advanced	Critical	Resolved
IDX-002	Drained Native Token from bulkbuy() Function	Advanced	High	Resolved
IDX-003	Improper Native Token Handling	Advanced	Medium	Resolved
IDX-004	Centralized Control of State Variable	General	Medium	Resolved *
IDX-005	Transaction Ordering Dependence	General	Low	Acknowledged
IDX-006	Smart Contract with Unpublished Source Code	General	Low	Acknowledged
IDX-007	Improper Function Visibility	Best Practice	Info	No Security Impact

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

5. Detailed Findings Information

5.1. Invalid tokenSaleList Calculation in sell() Function

ID	IDX-001
Target	DAgoraMarketplace
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** After the sale transaction takes effect, the NFTs that will transfer to the buyer will be reduced by 1 due to the logic flaw in the contract. This results in the buyer receiving one less NFT despite the fact that they have paid for the full amount. **Likelihood: High** The mentioned scenario can occur whenever a buyer signs a signature to buy NFT and the seller confirms the sale transaction, executing the `sell()` function.
Status	Resolved The DAgora team has fixed this issue by correcting the calculation of the length of the `_metaAddress` state as suggested.

5.1.1. Description

The `sell()` function is used for selling NFT on the `DAgoraMarketplace` contract with the buyer's signed signature, and the `tokenSaleList` state variable stores the total amount of NFT to be sold. However, the miscalculation in the `sell()` function causes the last NFT from the `_metaAddress` state to be excluded from the `tokenSaleList` list. The buyer will not receive the last NFT as committed in the signature.

The `_metaAddress.length` is the total amount of NFT for sale. Since the `_metaAddress.length` is always reduced by 1 at line 1539. Thus, the last NFT in `tokenSaleList` will be excluded.

DagoraMarketplace.sol

```
1532  function sell(address[] memory _metaAddress, uint256[] memory _metaUint,
      address[] memory _buyByTokenAddresses, uint256[] memory _buyByAmounts, uint16
      _saleByTokenIndex, bytes memory _signature) public payable
      isUnuseSignature(_signature) {
1533      require(_isValidTime(_metaUint[0], _metaUint[1]), "DAgora Marketplace:
      Invalid time");
1534      require(_metaAddress.length >= 1, "DAgora Marketplace: Invalid Input");
1535      require(_metaAddress.length + 3 == _metaUint.length, "DAgora Marketplace:
      Invalid Input");
```

```
1536
1537        require(_isInitPackage(_buyByTokenAddresses[_saleByTokenIndex]), "DAgora
    Marketplace: Invalid package");
1538
1539        address[] memory tokenSaleList = new address[](_metaAddress.length - 1);
1540        uint256[] memory tokenIdList = new uint256[](_metaUint.length - 3);
1541
1542        for (uint256 i = 0; i < _metaAddress.length - 1; i++) {
1543            tokenSaleList[i] = _metaAddress[i];
1544            tokenIdList[i] = _metaUint[i + 3];
1545        }
1546        address buyer = _getSigner(_metaAddress, _metaUint, _buyByTokenAddresses,
    _buyByAmounts, _signature);
1547
1548        _disableSignature(_signature);
1549        _buyToken(tokenSaleList, tokenIdList,
    _buyByTokenAddresses[_saleByTokenIndex], _buyByAmounts[_saleByTokenIndex],
    msg.sender, buyer);
1550
1551        emit Sell(buyer, msg.sender, _metaAddress, _metaUint,
    _buyByTokenAddresses[_saleByTokenIndex], _buyByAmounts[_saleByTokenIndex]);
1552    }
```

Consequently, even if the buyer has paid the full amount, they will not receive the last NFT.

5.1.2. Remediation

Inspex suggests fixing the `tokenSaleList` state calculation by removing the `-1` from `_metaAddress.length - 1` at lines 1539 and 1542, for example:

DagoraMarketplace.sol

```
1532    function sell(address[] memory _metaAddress, uint256[] memory _metaUint,
        address[] memory _buyByTokenAddresses, uint256[] memory _buyByAmounts, uint16
        _saleByTokenIndex, bytes memory _signature) public payable
        isUnuseSignature(_signature) {
1533        require(_isValidTime(_metaUint[0], _metaUint[1]), "DAgora Marketplace:
        Invalid time");
1534        require(_metaAddress.length >= 1, "DAgora Marketplace: Invalid Input");
1535        require(_metaAddress.length + 3 == _metaUint.length, "DAgora Marketplace:
        Invalid Input");
1536
1537        require(_isInitPackage(_buyByTokenAddresses[_saleByTokenIndex]), "DAgora
        Marketplace: Invalid package");
1538
1539        address[] memory tokenSaleList = new address[](_metaAddress.length);
1540        uint256[] memory tokenIdList = new uint256[](_metaUint.length - 3);
1541
```

```
1542        for (uint256 i = 0; i < _metaAddress.length; i++) {
1543            tokenSaleList[i] = _metaAddress[i];
1544            tokenIdList[i] = _metaUint[i + 3];
1545        }
1546        address buyer = _getSigner(_metaAddress, _metaUint, _buyByTokenAddresses,
    _buyByAmounts, _signature);
1547
1548        _disableSignature(_signature);
1549        _buyToken(tokenSaleList, tokenIdList,
    _buyByTokenAddresses[_saleByTokenIndex], _buyByAmounts[_saleByTokenIndex],
    msg.sender, buyer);
1550
1551        emit Sell(buyer, msg.sender, _metaAddress, _metaUint,
    _buyByTokenAddresses[_saleByTokenIndex], _buyByAmounts[_saleByTokenIndex]);
1552    }
```

5.2. Drained Native Token from bulkbuy() Function

ID	IDX-002
Target	DAgoraMarketplace
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium Anyone can drain a native token from the contract, which is the claim fee and the market fee of the platform. This results in a revenue loss for the platform. **Likelihood:** High Anyone can use this function to drain the native token from the contract as long as there exists the native token in the contract.
Status	Resolved The DAgora team has fixed this issue by removing the ability to receive a native token from the function.

5.2.1. Description

The `DAgoraMarketplace` contract offers the user an option to buy multiple NFTs within one function call, `bulkbuy()`. The `bulkbuy()` function will simply iterate through the call data and call the `buy()` function on each iteration.

DagoraMarketplace.sol

```
1556 function bulkbuy(BulkBuyParam[] memory _params) external payable {
1557     for (uint8 i; i < _params.length; i++) {
1558         buy(_params[i].metaAddress, _params[i].metaUint,
     _params[i].saleByTokenAddresses, _params[i].saleByAmounts,
     _params[i].buyByTokenIndex, _params[i].signature);
1559     }
1560 }
```

The `buy()` function will check the validity of the supplied parameters and the signature. Then, call the `_buyToken()` function to transfer the associated tokens.

DagoraMarketplace.sol

```
1498 function buy(address[] memory _metaAddress, uint256[] memory _metaUint,
     address[] memory _saleByTokenAddresses, uint256[] memory _saleByAmounts, uint16
     _buyByTokenIndex, bytes memory _signature) public payable
     isUnuseSignature(_signature) {
```

```
1499      require(_isValidTime(_metaUint[0], _metaUint[1]), "DAgora Marketplace:
Invalid time");
1500      require(_metaAddress.length >= 2, "DAgora Marketplace: Invalid Input");
1501      require(_metaAddress.length + 2 == _metaUint.length, "DAgora Marketplace:
Invalid Input");
1502
1503      require(_isInitPackage(_saleByTokenAddresses[_buyByTokenIndex]), "DAgora
Marketplace: Invalid package");
1504
1505      address[] memory tokenSaleList = new address[](_metaAddress.length - 1);
1506      uint256[] memory tokenIdList = new uint256[](_metaUint.length - 3);
1507
1508      for (uint256 i = 0; i < _metaAddress.length - 1; i++) {
1509          tokenSaleList[i] = _metaAddress[i + 1];
1510          tokenIdList[i] = _metaUint[i + 3];
1511      }
1512
1513      if (_metaAddress[0] != address(0)) {
1514          require(_metaAddress[0] == msg.sender, "DAgora Marketplace: Only
reserve address can make this payment");
1515      }
1516
1517      // Avoid stack too deep
1518      address seller = _getSigner(_metaAddress, _metaUint, _saleByTokenAddresses,
_saleByAmounts, _signature);
1519      _disableSignature(_signature);
1520      _buyToken(tokenSaleList, tokenIdList,
_saleByTokenAddresses[_buyByTokenIndex], _saleByAmounts[_buyByTokenIndex],
seller, msg.sender);
1521
1522      emit Buy(msg.sender, seller, _metaAddress, _metaUint,
_saleByTokenAddresses[_buyByTokenIndex], _saleByAmounts[_buyByTokenIndex]);
1523  }
```

The `_buyToken()` function is an internal function that transfers the buying token to the contract, transfers the NFTs from the seller to the buyer, and then transfers the buying token from the contract to the seller. If the buying token is a native token, there will be a check on the `msg.value` in line 1363 in the `_buyToken()` function.

DagoraMarketplace.sol

```
1360  function _buyToken(address[] memory _nftAddresses, uint256[] memory _nftIds,
address _token, uint256 _amount, address _seller, address _buyer) internal {
1361      // get amount to contract
1362      if (_token == address(0)) {
1363          require(msg.value >= _amount, "DAgora Marketplace: Not enough
payment");
```

```
1364        } else {
1365            IERC20(_token).safeTransferFrom(_buyer, address(this), _amount);
1366        }
1367
1368        // transfer Royalty fee: buyer -> creator
1369        uint256 amountAfterSubMarketFee = _subMarketFee(_amount,
    _packageInfos[_token].claimFee, _packageInfos[_token].martketFee);
1370        uint256 amountAfterSubRoyaltyFee = _subRoyaltyFee(_nftAddresses, _token,
    amountAfterSubMarketFee);
1371
1372        // transfer NFT: seller -> buyer
1373        _safeTransferNFT(_nftAddresses, _nftIds, _seller, _buyer);
1374
1375        // transfer token: contract -> seller
1376        if (_token == address(0)) {
1377            payable(_seller).transfer(amountAfterSubRoyaltyFee);
1378        } else {
1379            IERC20(_token).safeTransfer(_seller, amountAfterSubRoyaltyFee);
1380        }
1381    }
```

The problem will occur when a user buys multiple orders and the buying tokens are native tokens. The user can send the `msg.value` on the `bulkbuy()` function as high as the most expensive order in a native token, instead of the sum of every order in a native token. The unpaid native token amount will be transferred from the contract itself to the seller. Therefore, the user can leverage this fault to drain a native token from the contract by buying and selling to themselves.

5.2.2. Remediation

Inspex suggests calculating the total amount of the required native token in advance and checking this value with the `msg.value` state to prevent the user from sending fewer tokens than the required amount.

DagoraMarketplace.sol

```
1556 function bulkbuy(BulkBuyParam[] memory _params) external payable {
1557     uint256 totalNativeToken;
1558     for (uint8 i; i < _params.length; i++) {
1559         if (_params[i].saleByTokenAddresses[_params[i].buyByTokenIndex] ==
     address(0))
1560             totalNativeToken +=
     _params[i].saleByAmounts[_params[i].buyByTokenIndex];
1561         buy(_params[i].metaAddress, _params[i].metaUint,
     _params[i].saleByTokenAddresses, _params[i].saleByAmounts,
     _params[i].buyByTokenIndex, _params[i].signature);
1562     }
1563     require(msg.value >= totalNativeToken, "DAgora Marketplace: Insufficient
     fund");
```

5.3. Improper Native Token Handling

ID	IDX-003
Target	DAgoraMarketplace
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High The seller will lose their NFTs in the sale order that uses a native token as a buying token and also lose their native tokens to pay the platform's fee, which will increase along with the price of NFTs. **Likelihood:** Low It's very unlikely that the seller will execute the `sell()` function to sell the order without advance payment from the buyer.
Status	Resolved The DAgora team has fixed this issue by removing the `payable` modifier from the `sell()` function.

5.3.1. Description

The `DAgoraMarketplace` contract allows the seller to accept a buying offer from any buyer by executing the `sell()` function with the buyer's signature that they have signed for a commitment to buy the NFTs.

To be more specific, the `sell()` function will validate the data in the buying offer, verify the buyer's signature, and then initiate the transfer process by calling the `_buyToken()` function.

DagoraMarketplace.sol

```
1532 function sell(address[] memory _metaAddress, uint256[] memory _metaUint,
     address[] memory _buyByTokenAddresses, uint256[] memory _buyByAmounts, uint16
     _saleByTokenIndex, bytes memory _signature) public payable
     isUnuseSignature(_signature) {
1533     require(_isValidTime(_metaUint[0], _metaUint[1]), "DAgora Marketplace:
     Invalid time");
1534     require(_metaAddress.length >= 1, "DAgora Marketplace: Invalid Input");
1535     require(_metaAddress.length + 3 == _metaUint.length, "DAgora Marketplace:
     Invalid Input");
1536
1537     require(_isInitPackage(_buyByTokenAddresses[_saleByTokenIndex]), "DAgora
     Marketplace: Invalid package");
1538
1539     address[] memory tokenSaleList = new address[](_metaAddress.length - 1);
```

```
1540        uint256[] memory tokenIdList = new uint256[](_metaUint.length - 3);
1541
1542        for (uint256 i = 0; i < _metaAddress.length - 1; i++) {
1543            tokenSaleList[i] = _metaAddress[i];
1544            tokenIdList[i] = _metaUint[i + 3];
1545        }
1546        address buyer = _getSigner(_metaAddress, _metaUint, _buyByTokenAddresses,
    _buyByAmounts, _signature);
1547
1548        _disableSignature(_signature);
1549        _buyToken(tokenSaleList, tokenIdList,
    _buyByTokenAddresses[_saleByTokenIndex], _buyByAmounts[_saleByTokenIndex],
    msg.sender, buyer);
1550
1551        emit Sell(buyer, msg.sender, _metaAddress, _metaUint,
    _buyByTokenAddresses[_saleByTokenIndex], _buyByAmounts[_saleByTokenIndex]);
1552    }
```

After that, the `_buyToken()` function will transfer the buying token from the buyer to the contract, and then the contract will collect the platform's fee and transfer the rest to the seller. Finally, the contract will transfer the NFTs from the seller to the buyer.

If the buying token is a native token (`_token == address(0)`), the contract will look up the paid amount in the `msg.value` state, which is the value supplied by the transaction initiator: the seller.

DagoraMarketplace.sol

```
1360    function _buyToken(address[] memory _nftAddresses, uint256[] memory _nftIds,
    address _token, uint256 _amount, address _seller, address _buyer) internal {
1361        // get amount to contract
1362        if (_token == address(0)) {
1363            require(msg.value >= _amount, "DAgora Marketplace: Not enough
    payment");
1364        } else {
1365            IERC20(_token).safeTransferFrom(_buyer, address(this), _amount);
1366        }
1367
1368        // transfer Royalty fee: buyer -> creator
1369        uint256 amountAfterSubMarketFee = _subMarketFee(_amount,
    _packageInfos[_token].claimFee, _packageInfos[_token].martketFee);
1370        uint256 amountAfterSubRoyaltyFee = _subRoyaltyFee(_nftAddresses, _token,
    amountAfterSubMarketFee);
1371
1372        // transfer NFT: seller -> buyer
1373        _safeTransferNFT(_nftAddresses, _nftIds, _seller, _buyer);
1374
1375        // transfer token: contract -> seller
1376        if (_token == address(0)) {
```

```
1377            payable(_seller).transfer(amountAfterSubRoyaltyFee);
1378        } else {
1379            IERC20(_token).safeTransfer(_seller, amountAfterSubRoyaltyFee);
1380        }
1381    }
```

Therefore, if the buying offer has been defined to use the native token as the buy token, and the seller executes the `sell()` function to accept this buying offer, the seller has to pay the native tokens in advance to let the execution be successful, and there is no guarantee that the buyer will pay the advance payment back from the seller.

As a result, the contract should not permit the user to use the native token for the `sell()` function since the native token does not support allowance mechanisms like ERC20 tokens.

5.3.2. Remediation

Inspex suggests removing the `payable` modifier from the `sell()` function to disable the usage of a native token as a buy token.

DagoraMarketplace.sol

```
1532  function sell(address[] memory _metaAddress, uint256[] memory _metaUint,
      address[] memory _buyByTokenAddresses, uint256[] memory _buyByAmounts, uint16
      _saleByTokenIndex, bytes memory _signature) public isUnuseSignature(_signature)
      {
1533      require(_isValidTime(_metaUint[0], _metaUint[1]), "DAgora Marketplace:
      Invalid time");
1534      require(_metaAddress.length >= 1, "DAgora Marketplace: Invalid Input");
1535      require(_metaAddress.length + 3 == _metaUint.length, "DAgora Marketplace:
      Invalid Input");
1536
1537      require(_isInitPackage(_buyByTokenAddresses[_saleByTokenIndex]), "DAgora
      Marketplace: Invalid package");
1538
1539      address[] memory tokenSaleList = new address[](_metaAddress.length - 1);
1540      uint256[] memory tokenIdList = new uint256[](_metaUint.length - 3);
1541
1542      for (uint256 i = 0; i < _metaAddress.length - 1; i++) {
1543          tokenSaleList[i] = _metaAddress[i];
1544          tokenIdList[i] = _metaUint[i + 3];
1545      }
1546      address buyer = _getSigner(_metaAddress, _metaUint, _buyByTokenAddresses,
      _buyByAmounts, _signature);
1547
1548      _disableSignature(_signature);
1549      _buyToken(tokenSaleList, tokenIdList,
      _buyByTokenAddresses[_saleByTokenIndex], _buyByAmounts[_saleByTokenIndex],
      msg.sender, buyer);
```

```
1550
1551        emit Sell(buyer, msg.sender, _metaAddress, _metaUint,
    _buyByTokenAddresses[_saleByTokenIndex], _buyByAmounts[_saleByTokenIndex]);
1552    }
```

5.4. Centralized Control of State Variable

ID	IDX-004
Target	DAgoraMarketplace
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The controlling authorities can change the critical state variables. Thus, it is unfair to the other users. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	Resolved * The DAgora team has mitigated this issue by confirming that they will use the multi-sig account as an authorized party to ensure that all privileged actions are properly controlled, since the multi-sig account's execution requires that a list of members in the authorized party must agree.

5.4.1. 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:1374)	setRoyaltyFeeAdmin()	onlyAdmin
DagoraMarketplace.sol (L:1389)	changePaymentToken	onlyOwner
DagoraMarketplace.sol (L:1394)	registerPackage()	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

5.4.2. 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 others instead of `onlyAdmin` modifier to prevent it from being delayed.

5.5. Transaction Ordering Dependence

ID	IDX-005
Target	DAgoraMarketplace
Category	General Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** Low **Impact:** Low The registered owner of `RoyaltyFee` collection can instantly raise the royalty fee during the purchase from the buyers to reach the `ROYALTY_FEE_CAP` value, resulting in a lower received amount for the seller of the affected NFT collection. **Likelihood:** Medium Only the owner of the `RoyaltyFee` collection and an admin of the `DAgoraMarketplace` contract can set the royalty fee of each collection.
Status	**Acknowledged** The DAgora team has acknowledged and accepted the issue's risk and confirmed that the royalty fee is capped at 20%. Since the DAgora team has delegated the collection's owner right to their customer to update their royalty fee, If they update royalty fee without noticing anything to their users, that action may damage their reputation and revenue in the long run.

5.5.1. 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 20 percent.

DagoraMarketplace.sol

```
1037 uint constant PERCENT = 10000;
1038 uint constant ROYALTY_FEE_CAP = 2000;
```

DagoraMarketplace.sol

```
1400 function setRoyaltyFeeOwner(address _collection, bytes32 _byteCodeHash, uint256
     _nonce, bool _isCreate2,  uint32 _fee, uint256 _expiresAt) external override {
1401     require(_fee < ROYALTY_FEE_CAP, "DAgora Marketplace: Invalid input");
1402     uint256 royaltyFeeId = uint256(uint160(_collection));
1403     address owner = _getOwnerOfToken(DAgoraRoyaltyFee, royaltyFeeId);
```

```
1404        if (owner == address(0)) {
1405            require(_collection == _getContractAddress(_byteCodeHash, _nonce,
    _isCreate2), "DAgora Marketplace: Not creator of collection");
1406            IDAgora(DAgoraRoyaltyFee).mint(msg.sender, royaltyFeeId);
1407        } else {
1408            require(_getOwnerOfToken(DAgoraRoyaltyFee, royaltyFeeId) == msg.sender,
    "DAgora Marketplace: Not owner of royalty fee");
1409        }
1410
1411        _royaltyFeeConfigs[royaltyFeeId].fee = _fee;
1412        _royaltyFeeConfigs[royaltyFeeId].expiresAt = _expiresAt;
1413        emit UpdateRoyaltyFee(_collection, _fee, _expiresAt);
1414    }
```

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 through the `_subRoyaltyFee()` function in the `_buyToken()` function at line 1520.

DagoraMarketplace.sol

```
1498    function buy(address[] memory _metaAddress, uint256[] memory _metaUint,
        address[] memory _saleByTokenAddresses, uint256[] memory _saleByAmounts, uint16
        _buyByTokenIndex, bytes memory _signature) public payable
        isUnuseSignature(_signature) {
1499        require(_isValidTime(_metaUint[0], _metaUint[1]), "DAgora Marketplace:
        Invalid time");
1500        require(_metaAddress.length >= 2, "DAgora Marketplace: Invalid Input");
1501        require(_metaAddress.length + 2 == _metaUint.length, "DAgora Marketplace:
        Invalid Input");
1502
1503        require(_isInitPackage(_saleByTokenAddresses[_buyByTokenIndex]), "DAgora
        Marketplace: Invalid package");
1504
1505        address[] memory tokenSaleList = new address[](_metaAddress.length - 1);
1506        uint256[] memory tokenIdList = new uint256[](_metaUint.length - 3);
1507
1508        for (uint256 i = 0; i < _metaAddress.length - 1; i++) {
1509            tokenSaleList[i] = _metaAddress[i + 1];
1510            tokenIdList[i] = _metaUint[i + 3];
1511        }
1512
1513        if (_metaAddress[0] != address(0)) {
1514            require(_metaAddress[0] == msg.sender, "DAgora Marketplace: Only
        reserve address can make this payment");
1515        }
1516
1517        // Avoid stack too deep
1518        address seller = _getSigner(_metaAddress, _metaUint, _saleByTokenAddresses,
```

```
_saleByAmounts, _signature);
1519        _disableSignature(_signature);
1520        _buyToken(tokenSaleList, tokenIdList,
_saleByTokenAddresses[_buyByTokenIndex], _saleByAmounts[_buyByTokenIndex],
seller, msg.sender);
1521
1522        emit Buy(msg.sender, seller, _metaAddress, _metaUint,
_saleByTokenAddresses[_buyByTokenIndex], _saleByAmounts[_buyByTokenIndex]);
1523    }
```

DagoraMarketplace.sol

```
1360    function _buyToken(address[] memory _nftAddresses, uint256[] memory _nftIds,
address _token, uint256 _amount, address _seller, address _buyer) internal {
1361        // get amount to contract
1362        if (_token == address(0)) {
1363            require(msg.value >= _amount, "DAgora Marketplace: Not enough
payment");
1364        } else {
1365            IERC20(_token).safeTransferFrom(_buyer, address(this), _amount);
1366        }
1367
1368        // transfer Royalty fee: buyer -> creator
1369        uint256 amountAfterSubMarketFee = _subMarketFee(_amount,
_packageInfos[_token].claimFee, _packageInfos[_token].martketFee);
1370        uint256 amountAfterSubRoyaltyFee = _subRoyaltyFee(_nftAddresses, _token,
amountAfterSubMarketFee);
1371
1372        // transfer NFT: seller -> buyer
1373        _safeTransferNFT(_nftAddresses, _nftIds, _seller, _buyer);
1374
1375        // transfer token: contract -> seller
1376        if (_token == address(0)) {
1377            payable(_seller).transfer(amountAfterSubRoyaltyFee);
1378        } else {
1379            IERC20(_token).safeTransfer(_seller, amountAfterSubRoyaltyFee);
1380        }
1381    }
```

The deducted amount is based on the current royalty fee configuration as in lines 1314 and 1324.

DagoraMarketplace.sol

```
1307    function _subRoyaltyFee(address[] memory _collections, address _tokenAddress,
uint _amount) private returns(uint256) {
1308        uint256 subAmount = _amount;
1309        uint256 totalRoyaltyFee = 0;
1310
```

```
1311        if (_isSameCollection(_collections)) {
1312            uint256 royaltyFeeId = uint256(uint160(_collections[0]));
1313            if (_isValidateRoyaltyFee(royaltyFeeId)) {
1314                totalRoyaltyFee =
    subAmount.mul(_royaltyFeeConfigs[royaltyFeeId].fee).div(PERCENT);
1315                _transferRoyaltyFee(_collections[0], _tokenAddress,
    totalRoyaltyFee, _royaltyFeeConfigs[royaltyFeeId].fee);
1316            }
1317        } else {
1318            totalRoyaltyFee =
    subAmount.mul(_packageInfos[_tokenAddress].totalRoyaltyFee).div(PERCENT);
1319            uint256 totalPercent = 0;
1320
1321            for (uint i = 0; i < _collections.length; i++) {
1322                uint256 royaltyFeeId = uint256(uint160(_collections[i]));
1323                if (_isValidateRoyaltyFee(royaltyFeeId)) {
1324                    totalPercent =
    totalPercent.add(_royaltyFeeConfigs[royaltyFeeId].fee);
1325                }
1326            }
1327
1328            if (totalPercent == 0) {
1329                return _amount;
1330            }
1331
1332            for (uint i = 0; i < _collections.length; i++) {
1333                if (!_isTransferRoyaltyFee[_collections[i]]) {
1334                    _transferRoyaltyFee(_collections[i], _tokenAddress,
    totalRoyaltyFee, totalPercent);
1335                }
1336                _isTransferRoyaltyFee[_collections[i]] = true;
1337            }
1338            _clearTempTransferRoyaltyFee(_collections);
1339        }
1340
1341
1342        return _amount.sub(totalRoyaltyFee);
1343    }
```

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.

5.5.2. 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 the following option can be used to mitigate 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 expires by renewing the royalty fee configuration again.

5.6. Smart Contract with Unpublished Source Code

ID	IDX-006
Target	DAgoraMarketplace
Category	General Smart Contract Vulnerability
CWE	CWE-1006: Bad Coding Practices
Risk	**Severity:** Low **Impact:** Medium 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. **Likelihood:** Low 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	**Acknowledged** The DAgora 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. Improper Function Visibility

ID	IDX-007
Target	DAgoraMarketplace
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact The DAgora team has partially fixed this issue. There are two functions that have not changed the visibility into `external`.

5.7.1. Description

Public functions that are never called internally by the contract itself should have external visibility. This improves the readability of the contract, allowing clear distinction between functions that are externally used and functions that are also called internally.

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

DagoraMarketplace.sol

```
1532 function sell(address[] memory _metaAddress, uint256[] memory _metaUint,
     address[] memory _buyByTokenAddresses, uint256[] memory _buyByAmounts, uint16
     _saleByTokenIndex, bytes memory _signature) public payable
     isUnuseSignature(_signature) {
1533     require(_isValidTime(_metaUint[0], _metaUint[1]), "DAgora Marketplace:
     Invalid time");
1534     require(_metaAddress.length >= 1, "DAgora Marketplace: Invalid Input");
1535     require(_metaAddress.length + 3 == _metaUint.length, "DAgora Marketplace:
     Invalid Input");
1536
1537     require(_isInitPackage(_buyByTokenAddresses[_saleByTokenIndex]), "DAgora
     Marketplace: Invalid package");
1538
1539     address[] memory tokenSaleList = new address[](_metaAddress.length - 1);
1540     uint256[] memory tokenIdList = new uint256[](_metaUint.length - 3);
1541
1542     for (uint256 i = 0; i < _metaAddress.length - 1; i++) {
1543         tokenSaleList[i] = _metaAddress[i];
```

```
1544            tokenIdList[i] = _metaUint[i + 3];
1545        }
1546        address buyer = _getSigner(_metaAddress, _metaUint, _buyByTokenAddresses,
    _buyByAmounts, _signature);
1547
1548        _disableSignature(_signature);
1549        _buyToken(tokenSaleList, tokenIdList,
    _buyByTokenAddresses[_saleByTokenIndex], _buyByAmounts[_saleByTokenIndex],
    msg.sender, buyer);
1550
1551        emit Sell(buyer, msg.sender, _metaAddress, _metaUint,
    _buyByTokenAddresses[_saleByTokenIndex], _buyByAmounts[_saleByTokenIndex]);
1552    }
```

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

Target	Contract	Function
DagoraMarketplace.sol (L:942)	DAgoraMarketplace	renounceOwnership()
DagoraMarketplace.sol (L:951)	DAgoraMarketplace	transferOwnership()
DagoraMarketplace.sol (L:979)	DAgoraMarketplace	addAdmin()
DagoraMarketplace.sol (L:985)	DAgoraMarketplace	removeAdmin()
DagoraMarketplace.sol (L:1532)	DAgoraMarketplace	sell()

5.7.2. Remediation

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

DagoraMarketplace.sol

```
1532    function sell(address[] memory _metaAddress, uint256[] memory _metaUint,
    address[] memory _buyByTokenAddresses, uint256[] memory _buyByAmounts, uint16
    _saleByTokenIndex, bytes memory _signature) external payable
    isUnuseSignature(_signature) {
1533        require(_isValidTime(_metaUint[0], _metaUint[1]), "DAgora Marketplace:
    Invalid time");
1534        require(_metaAddress.length >= 1, "DAgora Marketplace: Invalid Input");
1535        require(_metaAddress.length + 3 == _metaUint.length, "DAgora Marketplace:
    Invalid Input");
1536
1537        require(_isInitPackage(_buyByTokenAddresses[_saleByTokenIndex]), "DAgora
    Marketplace: Invalid package");
```

```
1538
1539        address[] memory tokenSaleList = new address[](_metaAddress.length - 1);
1540        uint256[] memory tokenIdList = new uint256[](_metaUint.length - 3);
1541
1542        for (uint256 i = 0; i < _metaAddress.length - 1; i++) {
1543            tokenSaleList[i] = _metaAddress[i];
1544            tokenIdList[i] = _metaUint[i + 3];
1545        }
1546        address buyer = _getSigner(_metaAddress, _metaUint, _buyByTokenAddresses,
    _buyByAmounts, _signature);
1547
1548        _disableSignature(_signature);
1549        _buyToken(tokenSaleList, tokenIdList,
    _buyByTokenAddresses[_saleByTokenIndex], _buyByAmounts[_saleByTokenIndex],
    msg.sender, buyer);
1550
1551        emit Sell(buyer, msg.sender, _metaAddress, _metaUint,
    _buyByTokenAddresses[_saleByTokenIndex], _buyByAmounts[_saleByTokenIndex]);
1552    }
```

6. Appendix

6.1. About Inspex

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	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

inspeX

CYBERSECURITY PROFESSIONAL SERVICE