NFT Launchpad V2

Smart Contract Audit Report
Prepared for Ancient8

Date Issued:	Apr 5, 2023
Project ID:	AUDIT2023010
Version:	v1.0
Confidentiality Level:	Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2023010
Version	v1.0
Client	Ancient8
Project	NFT Launchpad V2
Auditor(s)	Wachirawit Kanpanluk
Author(s)	Wachirawit Kanpanluk
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Apr 5, 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 **1**
1.1. Audit Result 1
1.2. Disclaimer 1
2. Project Overview **2**
2.1. Project Introduction 2
2.2. Scope 3
3. Methodology **4**
3.1. Test Categories 4
3.2. Audit Items 5
3.3. Risk Rating 7
4. Summary of Findings **8**
5. Detailed Findings Information **10**
5.1. Improper Condition Checking in the Purchase of NFTs 10
5.2. Insecure Randomness in the Purchase of NFTs 14
5.3. Lack of Whitelist Sale Time Check 21
5.4. Smart Contract with Unpublished Source Code 25
5.5. Outdated Compiler Version 26
5.6. Inexplicit Solidity Compiler Version 27
6. Appendix **28**
6.1. About Inspex 28

1. Executive Summary

As requested by Ancient8, Inspex team conducted an audit to verify the security posture of the NFT Launchpad V2 smart contracts on Mar 31, 2023. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of NFT Launchpad V2 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 medium, 3 low, and 2 info-severity issues. With the project team's prompt response, 1 medium, 1 low and 2 info-severity issues were resolved in the reassessment, while 2 low-severity issues were acknowledged by the team. Therefore, Inspex trusts that NFT Launchpad V2 smart contracts have sufficient protections to be safe for public use. However, as the source code is not publicly available, the bytecode of the smart contracts deployed should be compared with the bytecode of the smart contracts audited before interacting with them. 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

Ancient8 - NFTLaunchpad is a contract that can be used to create an NFT box, which is the NFT contract. It allows the platform's users to participate in buying the NFT.

The NFT box has three types of sale times: private sale, public sale, and combination sale. The private and combination sale times have the whitelist who has a chance to buy before the public users. Public users are allowed to buy after that.

Scope Information:

Project Name	NFT Launchpad V2
Website	https://ancient8.gg
Smart Contract Type	Ethereum Smart Contract
Chain	BNB Smart Chain
Programming Language	Solidity
Category	NFT, Launchpad

Audit Information:

Audit Method	Whitebox
Audit Date	Mar 31, 2023
Reassessment Date	Apr 5, 2023

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 smart contracts with the following bytecodes were audited and reassessed by Inspex in detail:

Initial Audit:

Contract	Bytecode SHA256 Hash
NFT	610277249fa56ff54df9dc1a1937cc3055273162e915bad017633ebe8449ae67
NFTLaunchpad	a1d24fae1df46a7d01a1da4c9145e423ebbd116a7d3aff15ca51c82271b826be

Reassessment Audit:

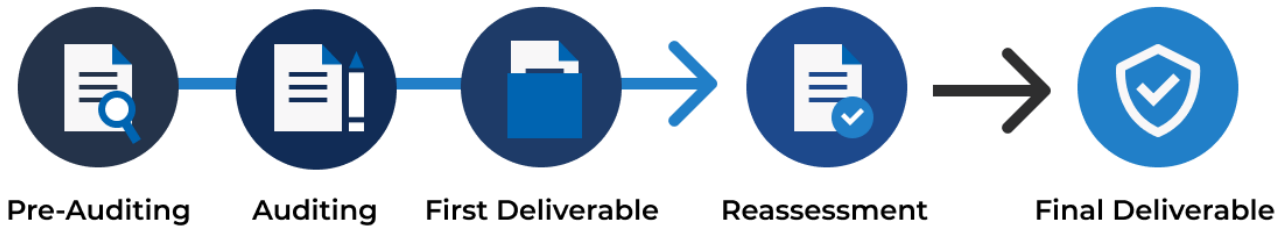
Contract	Bytecode SHA256 Hash
NFT	5553bca2f869b5f3f5ab1c65499f03245f118ecde6a503653bdbf5893bff687e
NFTLaunchpad	8bd201e8e2c5789ffcde508146be71c079af3fc4a3ad28af94c93830b3de3a9d

As the Ancient8 team has decided not to publish the source code to protect their intellectual property, users should compare the bytecode hashes, which were compiled by the 0.8.19 compiler version with no optimization, with the smart contracts deployed before interacting with them. This is to ensure that the contracts audited are the same as the ones being used.

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**.

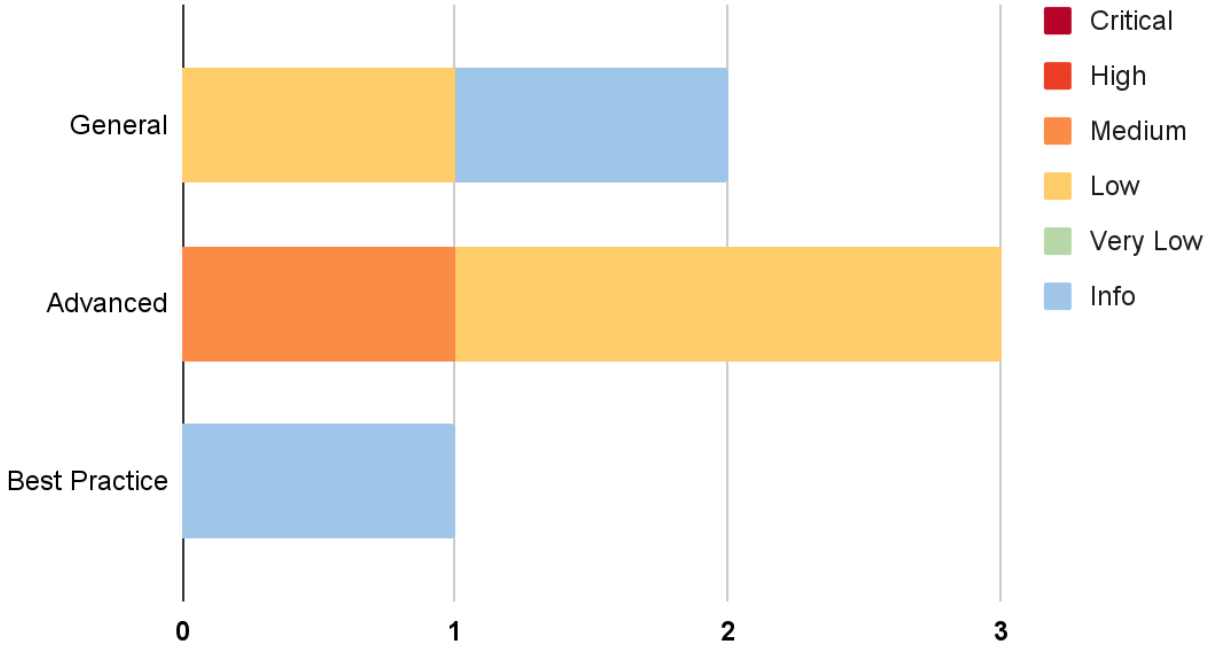
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would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

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

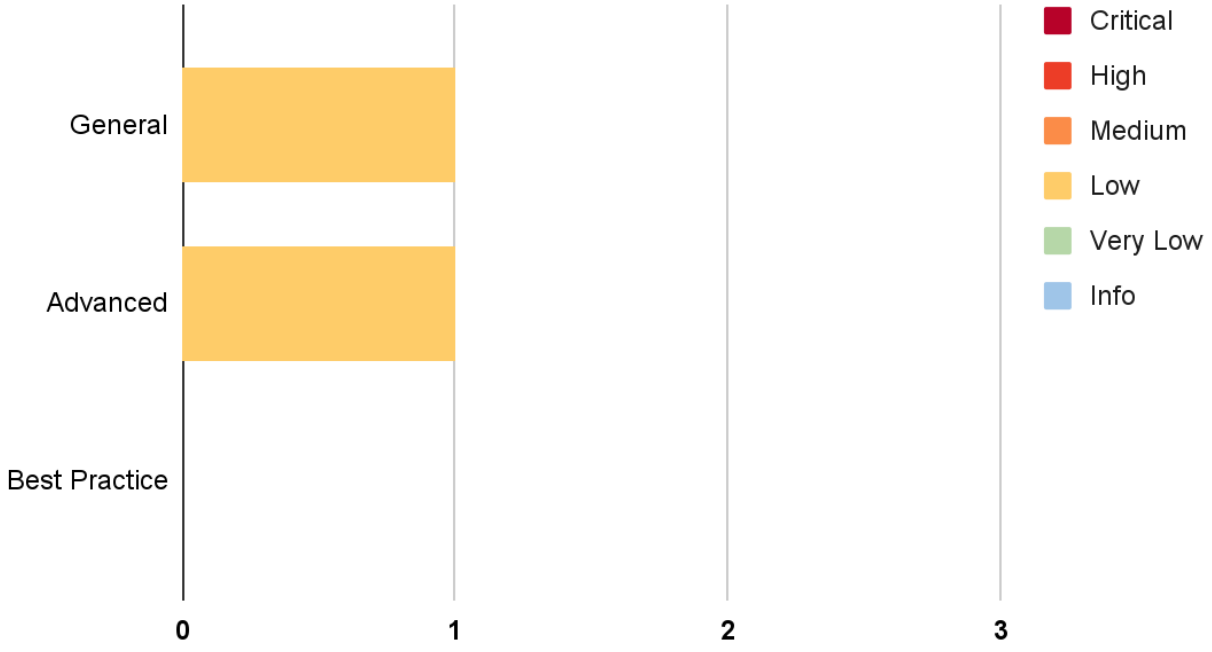
4. Summary of Findings

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	Improper Condition Checking in the Purchase of NFTs	Advanced	Medium	Resolved
IDX-002	Insecure Randomness in the Purchase of NFTs	Advanced	Low	Acknowledged
IDX-003	Lack of Whitelist Sale Time Check	Advanced	Low	Resolved
IDX-004	Smart Contract with Unpublished Source Code	General	Low	Acknowledged
IDX-005	Outdated Compiler Version	General	Info	Resolved
IDX-006	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved

* The mitigations or clarifications by Ancient8 can be found in Chapter 5.

5. Detailed Findings Information

5.1. Improper Condition Checking in the Purchase of NFTs

ID	IDX-001
Target	NFT
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The attacker can purchase NFTs exceeding the `purchaseLimit` state, especially for the whitelist sale. This results in unfairness to the other users. **Likelihood:** Medium It is likely to occur, especially with the whitelist of users who can purchase before other users. However, the attacker still has to pay for purchasing the NFT, which could not guarantee a profit.
Status	Resolved The Ancient8 team has resolved this issue as suggested by implementing a proper condition check when the whitelist buys NFTs to prevent them from exceeding the `purchaseLimit` state.

5.1.1. Description

In the `NFT` contract, the `buyNft()` function is used to purchase NFTs. The limit is set by condition check that the `balanceOf(msg.sender)` is less than the `purchaseLimit` in line 143.

NFT.sol

```
117 function buyNft() external payable {
118   uint256 timestamp = block.timestamp;
119
120   if (saleType == SaleType.whiteListSale) {
121     require(timestamp >= whiteListSaleTime && timestamp < saleEndTime, 'NFT
    only can be bought after white list sale time');
122
123     bool withinWhiteList = isAddressInwhiteList(msg.sender);
124
125     require(withinWhiteList, 'Only white list user can buy white list NFT');
126   }
127
128   if (saleType == SaleType.publicSale) {
129     require(timestamp >= publicSaleTime && timestamp < saleEndTime, 'NFT only
```

```
can be bought after public sale time');
130   }
131
132   if (saleType == SaleType.combinationSale) {
133     require(timestamp >= whiteListSaleTime && timestamp < saleEndTime, 'NFT
only can be bought after white list sale time');
134
135     if (timestamp < publicSaleTime) {
136       bool withinWhiteList = isAddressInwhiteList(msg.sender);
137
138       require(withinWhiteList, 'Only white list user can buy white list NFT');
139     }
140   }
141
142   if (purchaseLimit > 0) {
143       require(balanceOf(msg.sender) < purchaseLimit, 'User purchase amount
should not exceed purchase limit');
144   }
145
146   require(purchasedAmount < totalSupply, 'Purchased amount should be smaller
than total supply');
147
148   require(msg.value == nftPrice, 'Input price should be equal to NFT price');
149
150   uint randomIndex = random(tokenIds.length);
151   uint randomTokenId = tokenIds[randomIndex];
152
153   _transfer(address(this), msg.sender, randomTokenId);
154
155   payable(publisherAddress).transfer(msg.value);
156
157   emit BuyNft(randomTokenId, msg.sender);
158
159   tokenIds[randomIndex] = tokenIds[tokenIds.length - 1];
160   tokenIds.pop();
161
162   purchasedAmount += 1;
163 }
```

However, the `balanceOf(msg.sender)` can be reduced by transferring the NFT to another wallet or contract.

This provides an opportunity for the attacker to buy the NFT and exceed the `purchaseLimit` state, resulting in unfairness to the other users.

5.1.2. Remediation

Inspex suggests implementing a proper condition check when the whitelist buys NFTs to prevent them from

exceeding the `purchaseLimit` state, for example:

NFT.sol

```
117 mapping (address => uint256) whiteListPurchase;
118
119 function buyNft() external payable {
120   uint256 timestamp = block.timestamp;
121
122   if (saleType == SaleType.whiteListSale) {
123     require(timestamp >= whiteListSaleTime && timestamp < saleEndTime, 'NFT
    only can be bought after white list sale time');
124
125     bool withinWhiteList = isAddressInwhiteList(msg.sender);
126
127     require(withinWhiteList, 'Only white list user can buy white list NFT');
128   }
129
130   if (saleType == SaleType.publicSale) {
131     require(timestamp >= publicSaleTime && timestamp < saleEndTime, 'NFT only
    can be bought after public sale time');
132   }
133
134   if (saleType == SaleType.combinationSale) {
135     require(timestamp >= whiteListSaleTime && timestamp < saleEndTime, 'NFT
    only can be bought after white list sale time');
136
137     if (timestamp < publicSaleTime) {
138       bool withinWhiteList = isAddressInwhiteList(msg.sender);
139
140       require(withinWhiteList, 'Only white list user can buy white list NFT');
141     }
142   }
143
144   if (purchaseLimit > 0) {
145       require(whiteListPurchase[msg.sender] < purchaseLimit, 'User purchase
    amount should not exceed purchase limit');
146   }
147
148   require(purchasedAmount < totalSupply, 'Purchased amount should be smaller
    than total supply');
149
150   require(msg.value == nftPrice, 'Input price should be equal to NFT price');
151
152   whiteListPurchase[msg.sender] += 1;
153
154   uint randomIndex = random(tokenIds.length);
155   uint randomTokenId = tokenIds[randomIndex];
```

```
156
157     _transfer(address(this), msg.sender, randomTokenId);
158
159     payable(publisherAddress).transfer(msg.value);
160
161     emit BuyNft(randomTokenId, msg.sender);
162
163     tokenIds[randomIndex] = tokenIds[tokenIds.length - 1];
164     tokenIds.pop();
165
166     purchasedAmount += 1;
167 }
```

Please note that the remediations for other issues are not yet applied in the examples above.

5.2. Insecure Randomness in the Purchase of NFTs

ID	IDX-002
Target	NFT
Category	Advanced Smart Contract Vulnerability
CWE	CWE-330: Use of Insufficiently Random Values
Risk	**Severity:** Low **Impact:** Medium The attacker can buy a specific NFT to gain the valuable ones in the `buyNft()` functions. Thus, it is unfair to the other users and may cause a reputation loss to the platform. **Likelihood:** Low It is likely to occur since the `randomHash` can be predicted to obtain the favorable token ID. However, the value of the NFTs does not make this profitable because each NFT has nearly the same rarity. With low profits, there is low motivation.
Status	**Acknowledged** The Ancient8 team has acknowledged the issue, as they want to ensure a smooth purchase transaction for the user's experience.

5.2.1. Description

In the `NFT` contract, the `buyNft()` function is used to buy the NFT by transferring the native token, and the contract will transfer the NFT to the caller, as shown below in line 153.

NFT.sol

```
117 function buyNft() external payable {
118   uint256 timestamp = block.timestamp;
119
120   if (saleType == SaleType.whiteListSale) {
121     require(timestamp >= whiteListSaleTime && timestamp < saleEndTime, 'NFT
    only can be bought after white list sale time');
122
123     bool withinWhiteList = isAddressInwhiteList(msg.sender);
124
125     require(withinWhiteList, 'Only white list user can buy white list NFT');
126   }
127
128   if (saleType == SaleType.publicSale) {
129     require(timestamp >= publicSaleTime && timestamp < saleEndTime, 'NFT only
    can be bought after public sale time');
130   }
131
```

```
132   if (saleType == SaleType.combinationSale) {
133     require(timestamp >= whiteListSaleTime && timestamp < saleEndTime, 'NFT
only can be bought after white list sale time');
134
135     if (timestamp < publicSaleTime) {
136       bool withinWhiteList = isAddressInwhiteList(msg.sender);
137
138       require(withinWhiteList, 'Only white list user can buy white list NFT');
139     }
140   }
141
142   if (purchaseLimit > 0) {
143       require(balanceOf(msg.sender) < purchaseLimit, 'User purchase amount
should not exceed purchase limit');
144   }
145
146   require(purchasedAmount < totalSupply, 'Purchased amount should be smaller
than total supply');
147
148   require(msg.value == nftPrice, 'Input price should be equal to NFT price');
149
150   uint randomIndex = random(tokenIds.length);
151   uint randomTokenId = tokenIds[randomIndex];
152
153   _transfer(address(this), msg.sender, randomTokenId);
154
155   payable(publisherAddress).transfer(msg.value);
156
157   emit BuyNft(randomTokenId, msg.sender);
158
159   tokenIds[randomIndex] = tokenIds[tokenIds.length - 1];
160   tokenIds.pop();
161
162   purchasedAmount += 1;
163 }
```

The NFT that is transferred to the caller is defined by the `randomTokenId`, which is generated randomly from the `random()` function on line 150 by passing the length of the `tokenIds` array state to pick one of the indices in the state.

NFT.sol

```
109 function random(uint value) private view returns (uint) {
110   uint randomHash = uint(keccak256(abi.encodePacked(block.timestamp,
block.difficulty, blockhash(block.number - 1))));
111   return randomHash % value;
112 }
```

From the source code above, the `randomHash` value will be used to perform a modulo operation with the length that is passed to the function, and this will return a value for the `randomIndex`.

However, the `randomHash` can be predicted due to the use of the `block.timestamp`, `block.difficulty` and the blockhash of the previous block.

With the current design, the NFT that is provided by the contract is already revealed (`tokenURI`) in the `createNFT()` function as shown below:

NFT.sol

```
82  function createNFT(
83    string memory tokenURI,
84    address creatorAddress
85  ) external returns (uint256) {
86
87    require( owner == msg.sender, 'Only admin can create NFT');
88
89    _tokenId.increment();
90    uint256 newTokenId = _tokenId.current();
91
92    _mint(creatorAddress, newTokenId);
93    _setTokenURI(newTokenId, tokenURI);
94
95    _transfer(creatorAddress, address(this), newTokenId);
96
97    totalSupply += 1;
98    tokenIds.push(newTokenId);
99
100   emit CreateNft(newTokenId, creatorAddress);
101
102   return newTokenId;
103 }
```

This provides an opportunity for the caller to buy a specific token ID, resulting in unfairness to the other users.

5.2.2. Remediation

Inspex suggests redesigning the buying process to get the NFT token ID by using secure randomness, such as Chainlink VRF (https://docs.chain.link/docs/vrf/v2/introduction/), for example:

NFT.sol

```
1  import { VRFCoordinatorV2Interface } from
   "@chainlink/contracts/src/v0.8/interfaces/VRFCoordinatorV2Interface.sol";
2  import "@chainlink/contracts/src/v0.8/VRFConsumerBaseV2.sol";
3
```

```
4  contract NFT is VRFConsumerBaseV2, ERC721URIStorage, ReentrancyGuard {
```

NFT.sol

```
84  VRFCoordinatorV2Interface private vrfCoordinator;
85
86  // Your subscription ID.
87  uint64 s_subscriptionId; // Initital value in constructor
88  bytes32 keyHash; // Initital value in constructor
89  uint32 callbackGasLimit = 500000; // Initital value in constructor
90
91  // The default is 3, but you can set this higher.
92  uint16 requestConfirmations = 3; // Initital value in constructor
93
94  mapping(uint256 => address) buyAddress;
95
96  /**
97  * @notice buy NFT from the NFT Box
98  */
99  function buyNft() public payable {
100   require(tx.origin == msg.sender, "Can not call by Contract.");
101
102   uint256 timestamp = block.timestamp;
103
104   if (saleType == SaleType.whiteListSale) {
105     require(timestamp >= whiteListSaleTime && timestamp < saleEndTime, 'NFT
      only can be bought after white list sale time');
106
107     bool withinWhiteList = isAddressInwhiteList(msg.sender);
108
109     require(withinWhiteList, 'Only white list user can buy white list NFT');
110   }
111
112   if (saleType == SaleType.publicSale) {
113     require(timestamp >= publicSaleTime && timestamp < saleEndTime, 'NFT only
      can be bought after public sale time');
114   }
115
116   if (saleType == SaleType.combinationSale) {
117     require(timestamp >= whiteListSaleTime && timestamp < saleEndTime, 'NFT
      only can be bought after white list sale time');
118
119     if (timestamp < publicSaleTime) {
120       bool withinWhiteList = isAddressInwhiteList(msg.sender);
121
122       require(withinWhiteList, 'Only white list user can buy white list NFT');
123     }
124   }
```

```
125
126   if (purchaseLimit > 0) {
127       require(balanceOf(msg.sender) < purchaseLimit, 'User purchase amount
      should not exceed purchase limit');
128   }
129
130   require(purchasedAmount < totalSupply, 'Purchased amount should be smaller
      than total supply');
131
132   require(msg.value == nftPrice, 'Input price should be equal to NFT price');
133
134   payable(publisherAddress).transfer(msg.value);
135
136   emit BuyNft(randomTokenId, msg.sender);
137
138   purchasedAmount += 1;
139
140   uint256 requestId = vrfCoordinator.requestRandomWords(keyHash,
      vrfSubscriptionId, requestConfirmations, callbackGasLimit, 1);
141
142   buyAddress[requestId] = msg.sender;
143 }
144
145 /// @notice Chainlink's callback function to fulfill the randomness
146 /// @param _randomWords The random results from VRF
147 function fulfillRandomWords(
148     uint256 _requestId,
149     uint256[] memory _randomWords
150   ) internal override {
151     require(buyAddress[_requestId] != address(0), "Request not found");
152
153     uint256 randomIndex = _randomWords[0] % tokenIds.length;
154     uint256 randomTokenId = tokenIds[randomIndex];
155     address to = buyAddress[_requestId];
156
157     tokenIds[randomIndex] = tokenIds[tokenId.length - 1];
158     tokenIds.pop();
159     delete buyAddress[_requestId];
160
161     _transfer(address(this), to, randomTokenId);
162 }
```

Or using the blockhash of a future block as a random source, for example:

NFT.sol

```
109 struct RequestData {
110    uint256 randomBlock;
```

```
111     uint256 futureBlock;
112     address user;
113 }
114 mapping(uint256 => RequestData) request;
115 uint256 requestId = 0;
116 
117 function random(uint value, uint futureBlockNumber) private view returns (uint)
    {
118   uint randomHash =
    uint(keccak256(abi.encodePacked(blockhash(futureBlockNumber))));
119   return randomHash % value;
120 }
121 
122 /**
123 * @notice buy NFT from the NFT Box
124 */
125 function buyNft() external payable {
126   uint256 timestamp = block.timestamp;
127 
128   if (saleType == SaleType.whiteListSale) {
129     require(timestamp >= whiteListSaleTime && timestamp < saleEndTime, 'NFT
    only can be bought after white list sale time');
130 
131     bool withinWhiteList = isAddressInwhiteList(msg.sender);
132 
133     require(withinWhiteList, 'Only white list user can buy white list NFT');
134   }
135 
136   if (saleType == SaleType.publicSale) {
137     require(timestamp >= publicSaleTime && timestamp < saleEndTime, 'NFT only
    can be bought after public sale time');
138   }
139 
140   if (saleType == SaleType.combinationSale) {
141     require(timestamp >= whiteListSaleTime && timestamp < saleEndTime, 'NFT
    only can be bought after white list sale time');
142 
143     if (timestamp < publicSaleTime) {
144       bool withinWhiteList = isAddressInwhiteList(msg.sender);
145 
146       require(withinWhiteList, 'Only white list user can buy white list NFT');
147     }
148   }
149 
150   if (purchaseLimit > 0) {
151       require(balanceOf(msg.sender) < purchaseLimit, 'User purchase amount
    should not exceed purchase limit');
```

```
152   }
153
154   require(purchasedAmount < totalSupply, 'Purchased amount should be smaller
    than total supply');
155
156   require(msg.value == nftPrice, 'Input price should be equal to NFT price');
157
158   payable(publisherAddress).transfer(msg.value);
159
160   emit BuyNft(randomTokenId, msg.sender);
161
162   purchasedAmount += 1;
163
164   request[requestId] = RequestData(block.number, block.number + 1, msg.sender);
165   requestId += 1;
166 }
167
168 function claimNft(uint256 _requestId) external {
169   RequestData storage data = request[_requestId];
170
171   require(data.user != address(0), "Request not exists.");
172
173   if(block.number - data.futureBlock < 256){
174
175     uint randomIndex = random(tokenIds.length, data.futureBlock);
176     uint randomTokenId = tokenIds[randomIndex];
177
178     tokenIds[randomIndex] = tokenIds[tokenIds.length - 1];
179     tokenIds.pop();
180
181     _transfer(address(this), data.user, randomTokenId);
182   } else {
183     purchasedAmount -=1 ;
184   }
185
186   delete request[_requestId];
187 }
```

Please note that the remediations for other issues are not yet applied in the examples above.

5.3. Lack of Whitelist Sale Time Check

ID	IDX-003
Target	NFT
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Low **Impact:** Medium The whitelisted users will be unable to buy the NFT during the whitelist sale period. In order to solve this, the owner of the NFT contract will need to create a new contract which can result in a conflict between the old and new contracts. **Likelihood:** Low It is unlikely that the owner will intentionally or accidentally set the `whiteListSaleTime` state less than the `block.timestamp`.
Status	Resolved The Ancient8 team has resolved this issue as suggested by adding a validation check in the `constructor()` function to prevent the `_whiteListSaleTime` variable from being set to a value less than the `block.timestamp`.

5.3.1. Description

In the `NFT` contract, the `whiteListSaleTime`state will only be set once in the `constructor()` function as shown below in line 70.

NFT.sol

```
40  constructor(
41    uint256 _price,
42    uint _saleType,
43    uint256 _whiteListSaleTime,
44    uint256 _publicSaleTime,
45    uint256 _saleEendTime,
46    address _publisherAddress,
47    address _owner,
48    uint256 _purchaseLimit,
49    string memory _name,
50    string memory _symbol
51  ) ERC721(_name, _symbol) {
52    owner = payable(_owner);
53
54    nftPrice = _price;
55
```

```
56    if (_saleType == 0) {
57      saleType = SaleType.publicSale;
58      require(_publicSaleTime < _saleEendTime, "Public sale time must be less
      than sale end time.");
59    } else if (_saleType == 1) {
60      saleType = SaleType.whiteListSale;
61      require(_whiteListSaleTime < _saleEendTime, "White list sale time must be
      less than sale end time.");
62    } else if (_saleType == 2) {
63      saleType = SaleType.combinationSale;
64      require(_whiteListSaleTime < _publicSaleTime, "White list sale time must be
      less than public sale time.");
65      require(_publicSaleTime < _saleEendTime, "Public sale time must be less
      than sale end time.");
66    }
67
68    require(block.timestamp < _saleEendTime, "NFT box should be created before
      sale end time");
69
70    whiteListSaleTime = _whiteListSaleTime;
71    publicSaleTime = _publicSaleTime;
72    saleEndTime = _saleEendTime;
73    publisherAddress = _publisherAddress;
74    purchaseLimit = _purchaseLimit;
75  }
```

Then, the contract owner will use the `addWhiteList()` function to add whitelist addresses for the whitelist sale period before the `whiteListSaleTime` state is reached, as shown in line 195.

NFT.sol

```
190 function addWhiteList(address[] memory addresses) external {
191   require( owner == msg.sender, 'Only admin can add white list');
192   require( saleType == SaleType.whiteListSale || saleType ==
      SaleType.combinationSale, 'Only white list sale or combination sale supports
      adding white list');
193
194   uint256 timestamp = block.timestamp;
195   require(timestamp < whiteListSaleTime, 'Adding white list operation only can
      be executed before white list sale starts');
196
197   for (uint i = 0; i < addresses.length; i++) {
198       whiteListedAddresses[addresses[i]] = true;
199   }
200
201   emit AddWhiteList(addresses);
202 }
```

However, if the contract owner sets the `whiteListSaleTime` state to a value less than the current `block.timestamp`, the `addWhiteList()` function will be unable to be called due to the revert in line 195.

This means that the contract owner will have to create a new contract, or else the whitelist will be unable to buy the NFT during the whitelist sale period, which could lead to a conflict between the old and new contracts.

5.3.2. Remediation

Inspex suggests adding a validation check in the `constructor()` function to prevent the `_whiteListSaleTime` variable from being set to a value less than the `block.timestamp`, for example:

NFT.sol

```
40  constructor(
41    uint256 _price,
42    uint _saleType,
43    uint256 _whiteListSaleTime,
44    uint256 _publicSaleTime,
45    uint256 _saleEendTime,
46    address _publisherAddress,
47    address _owner,
48    uint256 _purchaseLimit,
49    string memory _name,
50    string memory _symbol
51  ) ERC721(_name, _symbol) {
52    owner = payable(_owner);
53
54    nftPrice = _price;
55
56    if(_saleType != 0) {
57      require(block.timestamp < _whiteListSaleTime, "White list sale time must be
      less than current time.");
58    }
59
60    if (_saleType == 0) {
61      saleType = SaleType.publicSale;
62      require(_publicSaleTime < _saleEendTime, "Public sale time must be less
      than sale end time.");
63    } else if (_saleType == 1) {
64      saleType = SaleType.whiteListSale;
65      require(_whiteListSaleTime < _saleEendTime, "White list sale time must be
      less than sale end time.");
66    } else if (_saleType == 2) {
67      saleType = SaleType.combinationSale;
68      require(_whiteListSaleTime < _publicSaleTime, "White list sale time must be
      less than public sale time.");
69      require(_publicSaleTime < _saleEendTime, "Public sale time must be less
```

```
than sale end time.");
70    }
71
72    require(block.timestamp < _saleEendTime, "NFT box should be created before
sale end time");
73
74    whiteListSaleTime = _whiteListSaleTime;
75    publicSaleTime = _publicSaleTime;
76    saleEndTime = _saleEendTime;
77    publisherAddress = _publisherAddress;
78    purchaseLimit = _purchaseLimit;
79  }
```

5.4. Smart Contract with Unpublished Source Code

ID	IDX-004
Target	NFT NFTLaunchpad
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 Ancient8 team has acknowledged this issue and decided not to publish the source code because the team wants to protect their intellectual property.

5.4.1. 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.

5.4.2. 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.

5.5. Outdated Compiler Version

ID	IDX-005
Target	NFT
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The Ancient8 team has resolved the issue as suggested by fixing the Solidity compiler version to 0.8.19.

5.5.1. Description

The Solidity compiler version specified in the `NFT` contract was outdated.

NFT.sol

```
1 // SPDX-License-Identifier: Apache-2.0
2 pragma solidity 0.8.17;
```

However, this version does not have publicly known inherent bugs that may potentially be used to cause damage to the smart contracts or the users of the smart contracts.

5.5.2. 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 0.8 is 0.8.19 (https://github.com/ethereum/solidity/releases), for example:

NFT.sol

```
1 // SPDX-License-Identifier: Apache-2.0
2 pragma solidity 0.8.19;
```

5.6. Inexplicit Solidity Compiler Version

ID	IDX-006
Target	NFTLaunchpad
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The Ancient8 team has resolved the issue as suggested by fixing the Solidity compiler version to 0.8.19.

5.6.1. 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, for example:

NftLaunchpad.sol

```
1 // SPDX-License-Identifier: Apache-2.0
2 pragma solidity ^0.8.17;
```

5.6.2. 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 0.8 is 0.8.19 (https://github.com/ethereum/solidity/releases), for example:

NftLaunchpad.sol

```
1 // SPDX-License-Identifier: Apache-2.0
2 pragma solidity 0.8.19;
```

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

inspeX

CYBERSECURITY PROFESSIONAL SERVICE