Launchpad (EVM)

Smart Contract Audit Report
Prepared for DAgora

Date Issued:	Aug 21, 2023
Project ID:	AUDIT2022051
Version:	v1.0
Confidentiality Level:	Public

Public

Report Information

Project ID	AUDIT2022051
Version	v1.0
Client	DAgora
Project	Launchpad (EVM)
Auditor(s)	Patipon Suwanbol Fungkiat Phadejtaku
Author(s)	Wachirawit Kanpanluk
Reviewer	Darunphop Pengkumta
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 Launchpad (EVM) smart contracts between Nov 1, 2022 and Nov 2, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Launchpad (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, 4 medium, 5 low, 2 very low, and 2 info-severity issues. With the project team's prompt response, 1 critical, 1 high, 4 medium, 1 low, 1 very low, and 1 info-severity issues were resolved or mitigated in the reassessment, while 4 low, 1 very low, and 1 info-severity issues were acknowledged by the team. Therefore, Inspex trusts that Launchpad (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 Launchpad is a project that allows the user who wants to have their launchpad contracts created to do so on their own. These launchpads are used to offer NFT redemption to their platforms' users. It also includes all necessary functions to support the business design for the launchpad creators. In exchange, there will be a fee collected for the DAgora.

Scope Information:

Project Name	Launchpad (EVM)
Website	https://dagora.xyz/
Smart Contract Type	Ethereum Smart Contract
Chain	BNB Smart Chain
Programming Language	Solidity
Category	NFT, Launchpad

Audit Information:

Audit Method	Whitebox
Audit Date	Nov 1, 2022 - Nov 2, 2022
Reassessment Date	Nov 14, 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
LaunchpadFactory	0b9550829aa2820b4af85d949a630bf8d7066912072ce9fe0c6a04d566495db0
MintableLaunchpad	3fbd3f7acd5f2bcdb9a8d5917d94e5ccec09af800a1393473ce6c9723e8002f9
TransferableLaunchpad	6d0752c6ebceb6b38149c8a952bab6dce88e3b09cc2f25860cf4069273a0bd95
AccessControl	420d259af82dae6af66a28c9921ee4a40b16db6cd44d50af1aa592e03ac54516
TimeLock	42f0a8445dd4d0c1375c037a03a21acf3a1ca042e337f0fa77db3d96ffe8171f

Reassessment

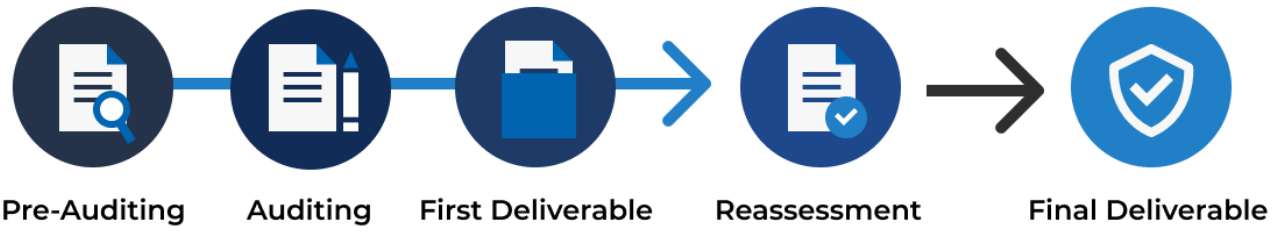
Contract	Bytecode SHA256 Hash
LaunchpadFactory	42341125c00da66b9c0a1e83e06d3ca6499e59229634330727fa879290f87744
MintableLaunchpad	8fb4d672eadfd154d77f042e3a85f31a448e22ae37f8aee20db775661a3c33fe
TransferableLaunchpad	bf044f83b2e63dcf20fadee4cae6892cc9094cd1e21c1f7a8ae163152d7c2580
AccessControl	420d259af82dae6af66a28c9921ee4a40b16db6cd44d50af1aa592e03ac54516
TimeLock	42f0a8445dd4d0c1375c037a03a21acf3a1ca042e337f0fa77db3d96ffe8171f

As the Coin98 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 0.8.9 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**.

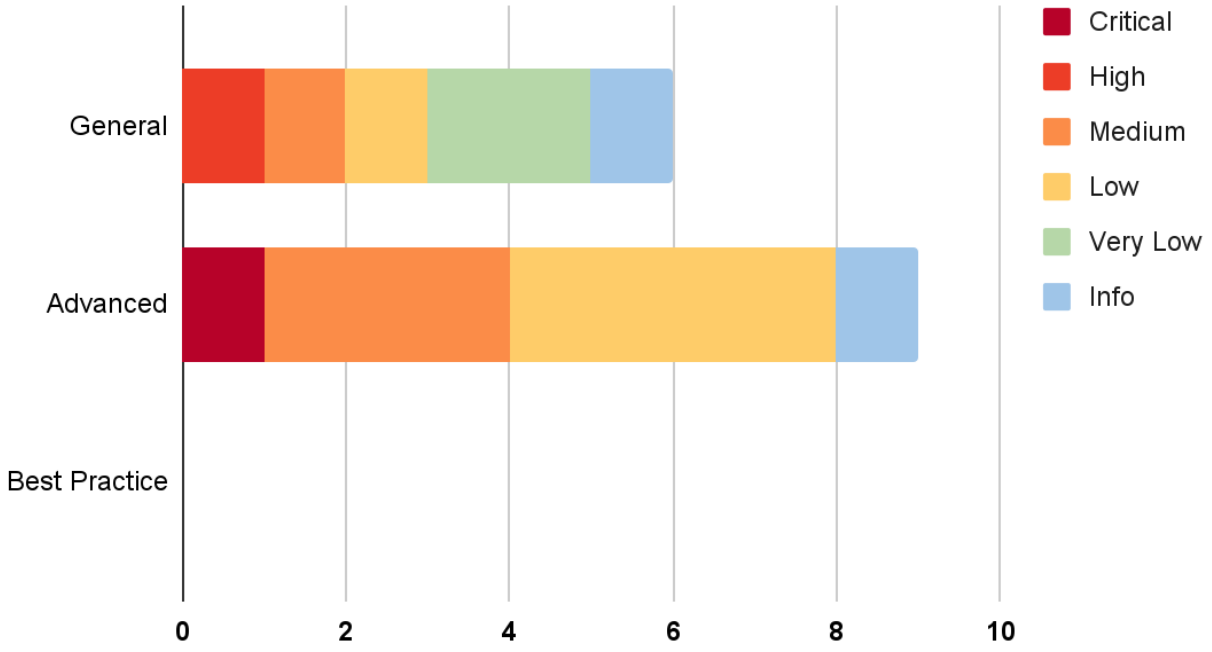
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would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

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

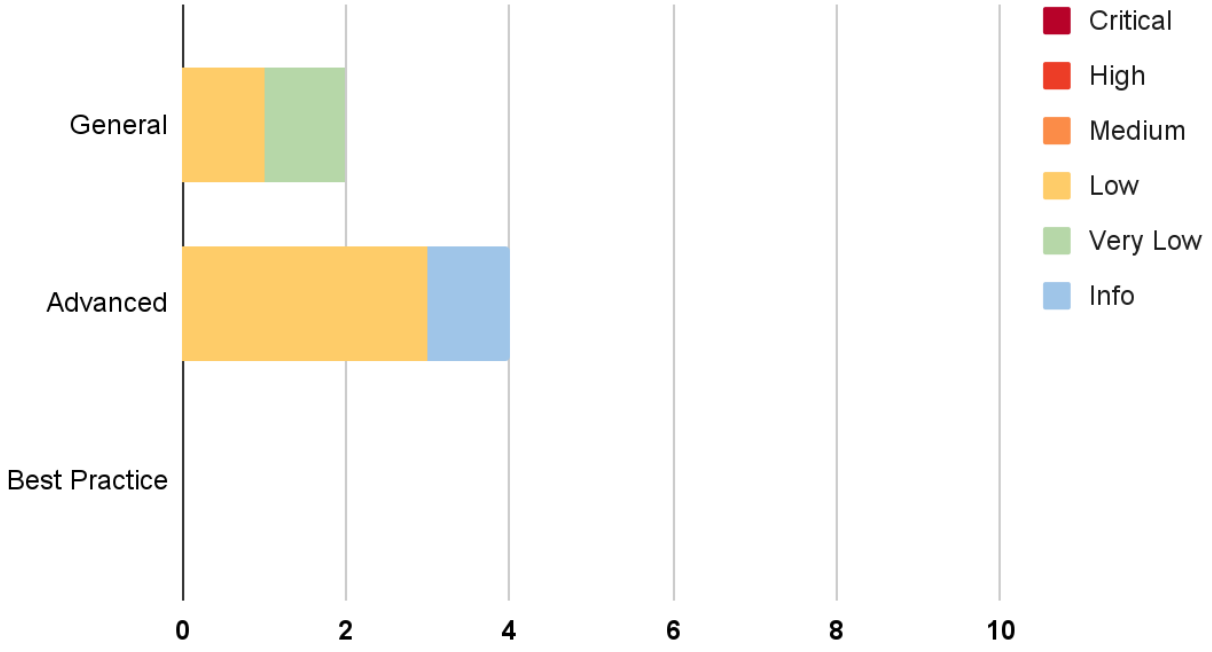
4. Summary of Findings

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	Insufficient Authorization for withdrawNft() Function	Advanced	Critical	Resolved
IDX-002	Reentrancy Attack in redeem() Function	General	High	Resolved
IDX-003	Centralized Control of State Variable	General	Medium	Resolved *
IDX-004	Missing Registered User Validation	Advanced	Medium	Resolved
IDX-005	Improper Access Control in setFee() function	Advanced	Medium	Resolved
IDX-006	Arbitrary Pre-Selection Index of Manual Minting	Advanced	Medium	Resolved
IDX-007	Withdrawable Converted NFT during Convert Time	Advanced	Low	Acknowledged
IDX-008	Insecure Randomness on Redeeming NFTs	Advanced	Low	Acknowledged
IDX-009	Improper Fee Enforcement	Advanced	Low	Resolved
IDX-010	Insufficient Parameter Validation in withdrawNft() Function	Advanced	Low	Acknowledged
IDX-011	Smart Contract with Unpublished Source Code	General	Low	Acknowledged
IDX-012	Outdated Compiler Version	General	Very Low	Acknowledged
IDX-013	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-014	Unchecked Return Value ERC20 Transfer	General	Info	Resolved
IDX-015	Insufficient Parameter Validation in redeem() Function	Advanced	Info	No Security Impact

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

5. Detailed Findings Information

5.1. Insufficient Authorization for withdrawNft() Function

ID	IDX-001
Target	MintableLaunchpad TransferableLaunchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: Critical** **Impact: High** Anyone can withdraw any NFTs that are stored in the contract at any time. This includes the valuable NFTs, which the platform will deposit later according to the business design. **Likelihood: High** When a valuable NFT is transferred to the contracts, any user will likely use the `withdrawNft()` function to acquire the deposited NFT since there is no cost preventing them from doing so except the gas.
Status	**Resolved** The Coin98 team has resolved this issue by adding `onlyOwner` modifier in the `withdrawNft()` functions to prevent anyone except the owner from withdrawing any NFT that are stored in the contract.

5.1.1. Description

In both `MintableLaunchpad` and `TransferableLaunchpad` contracts, it allows the launchpad's owner to store the NFT through the `pushToken()` function. The stored NFTs will be redeemed later by the launchpad users.

MintableLaunchpad.sol

```
68  function pushToken(uint256[] memory tokenIds) external onlyOwner {
69      require(address(_convertTokenAddress) != address(0), "Mintable Launchpad:
    Invalid convert token");
70
71      IERC721 token = IERC721(_convertTokenAddress);
72      for(uint256 i; i< tokenIds.length; i++) {
73          token.safeTransferFrom(_msgSender(), address(this), tokenIds[i]);
74          _tokenIds.push(tokenIds[i]);
75      }
76  }
```

TransferableLaunchpad.sol

```
32 function pushToken(uint256[] memory tokenIds) external onlyOwner {
33     IERC721 token = IERC721(_nftAddress);
34     for(uint256 i; i< tokenIds.length; i++) {
35         token.safeTransferFrom(_msgSender(), address(this), tokenIds[i]);
36         _tokenIds.push(tokenIds[i]);
37     }
38 }
```

However, there is the `withdrawNft()` function, allowing anyone to withdraw any NFTs in the `MintableLaunchpad` and `TransferableLaunchpad` contracts.

MintableLaunchpad.sol

```
98  function withdrawNft(address tokenAddress, uint256[] memory indexes) external {
99      if (tokenAddress == address(_convertTokenAddress)) {
100         for(uint i = 0; i < indexes.length; i++) {
101             uint256 tokenId = _tokenIds[indexes[i]];
102             _convertTokenAddress.safeTransferFrom(address(this), _msgSender(),
    tokenId);
103             _tokenIds[indexes[i]] = _tokenIds[_tokenIds.length - 1];
104             _tokenIds.pop();
105         }
106     } else {
107         IERC721 token = IERC721(tokenAddress);
108         for(uint i = 0; i < indexes.length; i++) {
109             token.safeTransferFrom(address(this), _msgSender(), indexes[i]);
110         }
111     }
112 }
```

TransferableLaunchpad.sol

```
40 function withdrawNft(address tokenAddress, uint256[] memory indexes) external {
41     if (tokenAddress == address(_nftAddress)) {
42         for(uint i = 0; i < indexes.length; i++) {
43             uint256 tokenId = _tokenIds[indexes[i]];
44             _nftAddress.safeTransferFrom(address(this), _msgSender(), tokenId);
45             _tokenIds[indexes[i]] = _tokenIds[_tokenIds.length - 1];
46             _tokenIds.pop();
47         }
48     } else {
49         IERC721 token = IERC721(tokenAddress);
50         for(uint i = 0; i < indexes.length; i++) {
51             token.safeTransferFrom(address(this), _msgSender(), indexes[i]);
52         }
53     }
54 }
```

As a result, anyone can withdraw any NFTs in the contracts. This includes the valuable NFTs that the platform owner will deposit for later usage.

5.1.2. Remediation

Inspex suggests adding an authorization to the `withdrawNft()` functions to suit the business design.

For example, applying the `onlyOwner` modifier to the `withdrawNft()` function, which has been already implemented in the contracts.

MintableLaunchpad.sol

```
98  function withdrawNft(address tokenAddress, uint256[] memory indexes) external
    onlyOwner {
99      if (tokenAddress == address(_convertTokenAddress)) {
100         for(uint i = 0; i < indexes.length; i++) {
101             uint256 tokenId = _tokenIds[indexes[i]];
102             _convertTokenAddress.safeTransferFrom(address(this), _msgSender(),
    tokenId);
103             _tokenIds[indexes[i]] = _tokenIds[_tokenIds.length - 1];
104             _tokenIds.pop();
105         }
106     } else {
107         IERC721 token = IERC721(tokenAddress);
108         for(uint i = 0; i < indexes.length; i++) {
109             token.safeTransferFrom(address(this), _msgSender(), indexes[i]);
110         }
111     }
112 }
```

TransferableLaunchpad.sol

```
40  function withdrawNft(address tokenAddress, uint256[] memory indexes) external
    onlyOwner {
41      if (tokenAddress == address(_nftAddress)) {
42          for(uint i = 0; i < indexes.length; i++) {
43              uint256 tokenId = _tokenIds[indexes[i]];
44              _nftAddress.safeTransferFrom(address(this), _msgSender(), tokenId);
45              _tokenIds[indexes[i]] = _tokenIds[_tokenIds.length - 1];
46              _tokenIds.pop();
47          }
48      } else {
49          IERC721 token = IERC721(tokenAddress);
50          for(uint i = 0; i < indexes.length; i++) {
51              token.safeTransferFrom(address(this), _msgSender(), indexes[i]);
52          }
53      }
54  }
```

5.2. Reentrancy Attack in redeem() Function

ID	IDX-002
Target	Launchpad
Category	General Smart Contract Vulnerability
CWE	CWE-841: Improper Enforcement of Behavioral Workflow
Risk	**Severity: High** **Impact: High** A Launchpad registered user can perform a reentrancy attack through the `redeem()` function. By doing this, the user can redeem more NFTs than the limit per user. **Likelihood: Medium** A Launchpad registered user address is required to be a contract address, which is unlikely to be settled by the owner of the launchpad contract. However, if the launchpad is in FCFS strategy (no whitelisted tree), there will be no restriction to prevent this scenario.
Status	**Resolved** The Coin98 team has resolved this issue by applying checks effect interactions to prevent reentrancy attacks.

5.2.1. Description

In the abstract `Launchpad` contract, it provides the `redeem()` function to allow the registered users to redeem the NFTs from the inheriting contracts, which are the `MintableLaunchpad` and `TransferableLaunchpad` contracts.

Launchpad.sol

```
146 function redeem(uint256 amount) onlyRedeemTime onlyRegister onlyActiveLaunchpad
    external payable {
147     require(_launchpadData.maxPerUser == 0 || _totalNftRedeemed[_msgSender()] +
    amount <= _launchpadData.maxPerUser, "Launchpad: Over max nft per user");
148     require(_launchpadData.maxRedeem == 0 || _totalRedeem + amount <=
    _launchpadData.maxRedeem, "Launchpad: Reach max redeem");
149     _claimFee(amount);
150
151     for (uint i = 0; i < amount; i++) {
152         _redeemToken();
153     }
154
155     _totalRedeem = _totalRedeem + uint32(amount);
156
157     _totalNftRedeemed[_msgSender()] = _totalNftRedeemed[_msgSender()] + amount;
```

```
158
159         emit Redeem(_msgSender(), amount);
160     }
```

For the `MintableLaunchpad` contract, the `_redeemToken()` function will mint the NFTs to the `msg.sender` as shown in the source code below.

MintableLaunchpad.sol

```
22  function _redeemToken() internal override {
23      uint256 currentIndex = totalSupply();
24      _safeMint(_msgSender(), currentIndex);
25  }
```

@openzeppelin/contracts-upgradeable/token/ERC721/ERC721Upgradeable.sol

```
260 function _safeMint(
261     address to,
262     uint256 tokenId,
263     bytes memory data
264 ) internal virtual {
265     _mint(to, tokenId);
266     require(
267         _checkOnERC721Received(address(0), to, tokenId, data),
268         "ERC721: transfer to non ERC721Receiver implementer"
269     );
270 }
```

For the `TransferableLaunchpad` contract, the `_redeemToken()` function will transfer the NFTs in the contract to the `msg.sender` as shown in the source code below.

TransferableLaunchpad.sol

```
18  function _redeemToken() internal override {
19      uint256 tokenIndex = _tokenIds.randomIndex(uint256(uint160(_msgSender())));
20      uint256 tokenId = _tokenIds[tokenIndex];
21
22      _nftAddress.safeTransferFrom(address(this), _msgSender(), tokenId);
23
24      _tokenIds[tokenIndex] = _tokenIds[_tokenIds.length - 1];
25      _tokenIds.pop();
26  }
```

@openzeppelin/contracts-upgradeable/token/ERC721/ERC721Upgradeable.sol

```
208 function _safeTransfer(
209         address from,
210         address to,
```

```
211         uint256 tokenId,
212         bytes memory data
213     ) internal virtual {
214         _transfer(from, to, tokenId);
215         require(_checkOnERC721Received(from, to, tokenId, data), "ERC721:
    transfer to non ERC721Receiver implementer");
216     }
```

In both scenarios, during NFT transfer, the `_checkOnERC721Received()` function will be executed, which will trigger the execution flow on that target address if it contains bytecode.

@openzeppelin/contracts-upgradeable/token/ERC721/ERC721Upgradeable.sol

```
399 function _checkOnERC721Received(
400     address from,
401     address to,
402     uint256 tokenId,
403     bytes memory data
404 ) private returns (bool) {
405     if (to.isContract()) {
406         try IERC721ReceiverUpgradeable(to).onERC721Received(_msgSender(), from,
    tokenId, data) returns (bytes4 retval) {
407             return retval ==
    IERC721ReceiverUpgradeable.onERC721Received.selector;
408         } catch (bytes memory reason) {
409             if (reason.length == 0) {
410                 revert("ERC721: transfer to non ERC721Receiver implementer");
411             } else {
412                 /// @solidity memory-safe-assembly
413                 assembly {
414                     revert(add(32, reason), mload(reason))
415                 }
416             }
417         }
418     } else {
419         return true;
420     }
421 }
```

As a result, the `redeem()` function is vulnerable to a reentrancy attack. For example, redeeming NFTs more than the limit per user due to non-applying "Checks Effects Interactions".

5.2.2. Remediation

Inspex suggests applying "Checks Effects Interactions" to the `redeem()` function or adding a `nonReentrant` modifier from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/security/ReentrancyGuardUpgradeable.sol).

For example, applying "Checks Effects Interactions":

Launchpad.sol

```
146 function redeem(uint256 amount) onlyRedeemTime onlyRegister onlyActiveLaunchpad
    external payable {
147     require(_launchpadData.maxPerUser == 0 || _totalNftRedeemed[_msgSender()] +
    amount <= _launchpadData.maxPerUser, "Launchpad: Over max nft per user");
148     require(_launchpadData.maxRedeem == 0 || _totalRedeem + amount <=
    _launchpadData.maxRedeem, "Launchpad: Reach max redeem");
149
150     _totalRedeem = _totalRedeem + uint32(amount);
151     _totalNftRedeemed[_msgSender()] = _totalNftRedeemed[_msgSender()] + amount;
152
153     _claimFee(amount);
154
155     for (uint i = 0; i < amount; i++) {
156         _redeemToken();
157     }
158
159     emit Redeem(_msgSender(), amount);
160 }
```

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

5.3. Centralized Control of State Variable

ID	IDX-003
Target	Launchpad LaunchpadFactory
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 to gain additional profit. 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 privileged roles.
Status	Resolved * The Coin98 team has mitigated this issue by adding a timelock mechanism for `setImplement()` function, but for `setLaunchpadStatus()` function they will use it for an emergency situation.

5.3.1. 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:

File	Contract	Function	Modifier
Launchpad.sol (L:120)	Launchpad	setLaunchpadStatus()	onlyOwner
LaunchpadFactory.sol (L:105)	LaunchpadFactory	setImplement()	onlyOwner

5.3.2. Remediation

Due to the business design, Inspex suggests applying a timelock mechanism to delay the changes for a reasonable amount of time e.g., 24 hours.

5.4. Missing Registered User Validation

ID	IDX-004
Target	Launchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The launchpad will miscount the actual number of the registered users if the user registers with the same `index` and `proofs` more than once, and this could lead to an attack by repeatedly registering with the same data until the counting number reaches the maximum registered number. **Likelihood:** Medium Only the users in the allowlist can perform the attack, and the attacker must pay fees which increase from the number of registrations. However, in a normal situation, the user can execute this function more than once.
Status	Resolved The Coin98 team has resolved this issue by adding validation for the registered user in the `register()` function.

5.4.1. Description

In the `Launchpad` contract, the allowlisted users can execute the `register()` function in the registration phase to claim their right to redeem the token in the redemption phase, and the users must provide the `index` and `proofs` values to prove that they are actually included in the allowlist.

Launchpad.sol

```
129 function register(uint256 index, bytes32[] memory proofs) onlyRegisterTime
    onlyActiveLaunchpad external {
130     if (_launchpadData.whitelistRoot != bytes32(0)) {
131         bytes32 leaf = keccak256(abi.encodePacked(index, _msgSender()));
132         require(MerkleProof.verify(proofs, _launchpadData.whitelistRoot, leaf),
    "Launchpad: not in whitelist");
133     }
134
135     require(_launchpadData.maxRegister == 0 || _totalRegister + 1 <=
    _launchpadData.maxRegister, "Launchpad: Reach max register");
136     ++_totalRegister;
137
138     _registerAddresses[_msgSender()] = true;
```

```
139
140         emit Register(_msgSender());
141     }
```

In the current implementation, there is no restriction to prevent the user from registering with the same proof data more than once, resulting in the `_totalRegister` not being the actual number of registered users because it increases along with the number of successful executions even if the user has been registered before.

Moreover, this can lead to the denial of service attack by repeatedly registering until the `_totalRegister` reaches the `_launchpadData.maxRegister`, at which point other users in the allowlist are not able to register anymore because of the miscounting of the registered users.

5.4.2. Remediation

Inspex suggests implementing a restriction condition to prevent the registered users from executing the `register()` function more than once as in the following source code:

Launchpad.sol

```
129 function register(uint256 index, bytes32[] memory proofs) onlyRegisterTime
    onlyActiveLaunchpad external {
130     require(_registerAddresses[_msgSender()] == false, "Launchpad: Already
    registered");
131
132     if (_launchpadData.whitelistRoot != bytes32(0)) {
133         bytes32 leaf = keccak256(abi.encodePacked(index, _msgSender()));
134         require(MerkleProof.verify(proofs, _launchpadData.whitelistRoot, leaf),
    "Launchpad: not in whitelist");
135     }
136
137     require(_launchpadData.maxRegister == 0 || _totalRegister + 1 <=
    _launchpadData.maxRegister, "Launchpad: Reach max register");
138     ++_totalRegister;
139
140     _registerAddresses[_msgSender()] = true;
141
142     emit Register(_msgSender());
143 }
```

5.5. Improper Access Control in setFee() function

ID	IDX-005
Target	Launchpad LaunchpadFactory MintableLaunchpad TransferableLaunchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The launchpad's owner can change the fee values to zero to avoid paying the protocol fee to the platform (the `LaunchpadFactory` contract), and the platform can not do anything about it. **Likelihood:** Medium Only the owner of the launchpad can execute the `setFee()` function.
Status	Resolved The Coin98 team has resolved this issue by adding `onlyFactoryOwner` modifier that can only set the protocol fee and separate the `setFee()` function to `setSharingFee()` and `setProtocolFee()`.

5.5.1. Description

In the `Launchpad` contract, the launchpad's owner can change the fee values, which are the `_sharingFee` and `_protocolFee` variables, by executing the `setFee()` function.

Launchpad.sol

```
110 function setFee(uint256 sharingFee, uint256 protocolFee) external override
    onlyOwner {
111     _sharingFee = sharingFee;
112     _protocolFee = protocolFee;
113
114     emit SetFee(sharingFee, protocolFee);
115 }
```

The fee values will affect the protocol fee while being calculated in the `_claimFee()` function.

Launchpad.sol

```
72  function _claimFee(uint256 total) internal {
73      uint256 feeRedeem = _launchpadData.feeRedeem * total;
74      if (feeRedeem > 0) {
75          address feeAddress = _factory.getFeeAddress();
76          uint256 protocolFee = (feeRedeem * _protocolFee / 10000) + _sharingFee;
77          if (_launchpadData.feeRedeemAddress == address(0)) {
78              require(msg.value >= feeRedeem, "Launchpad: Exceed fee");
79              (bool sent,) = payable(feeAddress).call{value: protocolFee}("");
80              require(sent, "Launchpad: Fail to send protocol fee");
81          } else {
82              IERC20 token = IERC20(_launchpadData.feeRedeemAddress);
83              token.safeTransferFrom(_msgSender(), address(this), feeRedeem);
84              token.safeTransfer(feeAddress, protocolFee);
85          }
86      }
87  }
```

Therefore, if the owner of the launchpad contract changes the `_sharingFee` and `_protocolFee` to 0, the result from the fee formula will be 0, and the platform will not receive any fee from the launchpad contract.

5.5.2. Remediation

Inspex suggests changing the visibility of the `setFee()` function from `external` to `internal`, and call this function in the `init()` function as in the following source code:

Launchpad.sol

```
110 function setFee(uint256 sharingFee, uint256 protocolFee) internal {
111     _sharingFee = sharingFee;
112     _protocolFee = protocolFee;
113
114     emit SetFee(sharingFee, protocolFee);
115 }
```

MintableLaunchpad.sol

```
30  function init(string memory name, string memory symbol, string memory baseUri,
    bool enableMinter, uint256 sharingFee, uint256 protocolFee) initializer
    external override {
31      __ERC721_init(name, symbol);
32      __Ownable_init();
33      _launchpadInit(_msgSender());
34
35      _enableMinter = enableMinter;
36      _baseUri = baseUri;
37      setFee(sharingFee, protocolFee);
38  }
```

TransferableLlaunchpad.sol

```
26 function init(address nftAddress, uint256 sharingFee, uint256 protocolFee)
   initializer override external {
27     _nftAddress = IERC721(nftAddress);
28     __Ownable_init();
29     _launchpadInit(_msgSender());
30     setFee(sharingFee, protocolFee);
31 }
```

For the `LaunchpadFactory` contract, the function calling of the `launchpad.setFee()` function has been moved to be called in the launchpad contracts.

LaunchpadFactory.sol

```
73 function createMintableLaunchpad(address owner, string memory name, string
   memory symbol, string memory baseUri, bool enableMinter,
   ILaunchpad.LaunchpadData memory data, uint256 sharingFee, uint256 protocolFee)
   external onlyCreator returns(address) {
74     address newLaunchpadAddress =
   _cloneLaunchpad(0x444f313053c893c305c4a5f333f3b033d548405c830016c4b623e787aa045
   145);
75     IMintableLaunchpad launchpad = IMintableLaunchpad(newLaunchpadAddress);
76
77     launchpad.init(name, symbol, baseUri, enableMinter, sharingFee,
   protocolFee);
78     launchpad.setLaunchpad(data);
79     // launchpad.setFee(sharingFee, protocolFee); Remove this line
80     Ownable(newLaunchpadAddress).transferOwnership(owner);
81
       emit CreateMintableLaunchpad(newLaunchpadAddress, owner, name, symbol,
82 enableMinter, data);
83     return newLaunchpadAddress;
84 }
85
86 /**
87  * @dev Create transferable launchpad.
88  */
   function createTransferableLaunchpad(address owner, address nftAddress,
   ILaunchpad.LaunchpadData memory data, uint256 sharingFee, uint256 protocolFee)
89 external onlyCreator returns(address) {
       address newLaunchpadAddress =
   _cloneLaunchpad(0x1bc7992855b26a5ae511e9b448c90941ef8f1b835f4936d94c8cfd9202da9
90 384);
       ITransferableLaunchpad launchpad =
91 ITransferableLaunchpad(newLaunchpadAddress);
```

```
92
93      launchpad.init(nftAddress, sharingFee, protocolFee);
94      launchpad.setLaunchpad(data);
95      // launchpad.setFee(sharingFee, protocolFee); Remove this line
96      Ownable(newLaunchpadAddress).transferOwnership(owner);
97
98      emit CreateTransferableLaunchpad(newLaunchpadAddress, owner, nftAddress,
    data);
99      return newLaunchpadAddress;
100 }
```

5.6. Arbitrary Pre-Selection Index of Manual Minting

ID	IDX-006
Target	MintableLaunchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The launchpad's owner can manually mint the desired index of NFT for anyone. By selecting an index to mint, the `_redeem()` function can result in being unusable. **Likelihood:** Medium Only the launchpad's owner can execute the `mint()` function. And, it is required that there must be at least one time that the `mint()` function is executed with the `tokenId` that surpasses the current index (total supply). Therefore, when the next `tokenId` to be minted is equal to that manual mint `tokenId`, this issue will occur.
Status	Resolved The Coin98 team has resolved this issue by changing the `mint()` function from using a specific `tokenId` for manually mint the NFT to using the `currentIndex` of `totalSupply()`

5.6.1. Description

In the `MintableLaunchpad` contract, in case of the contract has enabled the manual minting feature (`_enableMinter` is `true`), the launchpad's owner can mint an NFT by providing the destination address and the index of the new NFT.

MintableLaunchpad.sol

```
49 function mint(address owner, uint256 tokenId) external onlyOwner {
50     require(_enableMinter, "Mintable Launchpad: Not enable minter");
51
52     _safeMint(owner, tokenId);
53 }
```

However, manual minting will affect the total supply of the contract, which will be used in the normal minting during the redemption process.

```
22 function _redeemToken() internal override {
23     uint256 currentIndex = totalSupply();
24     _safeMint(_msgSender(), currentIndex);
25 }
```

As a result, manual minting with a pre-selected index can break the business logic.

For example, the current total supply is 5, and manual minting has been enabled.:

1. The launchpad's owner executes the `mint()` function by providing Bob's address as the owner address and the index of the new NFT as 100.
2. The current total supply was increased to 6.
3. The registered user executes the `redeem()` function and then receives the 6th NFT, which should be the 5th NFT.
4. The 5th NFT was skipped, and no one owns it.
5. If the other users keep redeeming the NFT until the total reaches 100, the transaction that transfers the 100th NFT will be reverted, because the owner of the 100th NFT is Bob now.

5.6.2. Remediation

Inspex suggests removing the `mint()` function from the `MintableLaunchpad` contract to prevent manually minting tokens by the owner.

However, if the `mint()` function is required to suit the business design, it is also suggested to allow only minting with the sequential `tokenId` instead of the selected `tokenId`. For example,

MintableLaunchpad.sol

```
49 function mint(address owner) external onlyOwner {
50     require(_enableMinter, "Mintable Launchpad: Not enable minter");
51
52     _safeMint(owner, totalSupply());
53 }
```

5.7. Withdrawable Converted NFT during Convert Time

ID	IDX-007
Target	MintableLaunchpad TransferableLaunchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The launchpad's owner can deposit valuable NFTs as prizes to convince the launchpad's user to pay a fee for the lucky draw. After that, the launchpad's owner can simply withdraw the valuable NFT and take the fee, so the launchpad's users pay a fee for nothing. **Likelihood:** Low It is unlikely that the launchpad's owner will execute the mentioned scenario since they are required to register with the platform first.
Status	**Acknowledged** The Coin98 team has acknowledged this issue by applying a condition to verify the phase of the launchpad. However, the validation can be bypassed by setting the `_convertTime` and `_convertEndTime` states to 0.

5.7.1. Description

In the `MintableLaunchpad` contract, the launchpad's owner can store NFTs through the `pushToken()` function. These NFTs are used as the Mystery Box, allowing the launchpad users to participate with a lucky draw.

MintableLaunchpad.sol

```
68  function pushToken(uint256[] memory tokenIds) external onlyOwner {
69      require(address(_convertTokenAddress) != address(0), "Mintable Launchpad:
    Invalid convert token");
70
71      IERC721 token = IERC721(_convertTokenAddress);
72      for(uint256 i; i< tokenIds.length; i++) {
73          token.safeTransferFrom(_msgSender(), address(this), tokenIds[i]);
74          _tokenIds.push(tokenIds[i]);
75      }
76  }
```

The launchpad's user must burn the NFT from the redemption phase in order to do a lucky draw through the `convertToken()` function.

MintableLaunchpad.sol

```
81  function convertToken(uint256 tokenId) external {
82      require(address(_convertTokenAddress) != address(0), "Mintable Launchpad:
    Not mistery box");
83      require(_convertTime <= block.timestamp, "Mintable Launchpad: Not convert
    time");
84      require(_tokenIds.length > 0, "Mintable Launchpad: Not enough token");
85      require(ownerOf(tokenId) == _msgSender(), "Mintable Launchpad: Not an
    owner");
86
87      _burn(tokenId);
88
89      uint256 tokenIndex = _tokenIds.randomIndex(tokenId);
90      uint256 receiveTokenId = _tokenIds[tokenIndex];
91
92      _convertTokenAddress.safeTransferFrom(address(this), _msgSender(),
    receiveTokenId);
93
94      _tokenIds[tokenIndex] = _tokenIds[_tokenIds.length - 1];
95      _tokenIds.pop();
96  }
```

However, the launchpad's owner can always withdraw the valuable NFTs that have been used as the prize for the launchpad's users through the `withdrawNft()` function.

MintableLaunchpad.sol

```
98  function withdrawNft(address tokenAddress, uint256[] memory indexes) external {
99      if (tokenAddress == address(_convertTokenAddress)) {
100         for(uint i = 0; i < indexes.length; i++) {
101             uint256 tokenId = _tokenIds[indexes[i]];
102             _convertTokenAddress.safeTransferFrom(address(this), _msgSender(),
    tokenId);
103             _tokenIds[indexes[i]] = _tokenIds[_tokenIds.length - 1];
104             _tokenIds.pop();
105         }
106     } else {
107         IERC721 token = IERC721(tokenAddress);
108         for(uint i = 0; i < indexes.length; i++) {
109             token.safeTransferFrom(address(this), _msgSender(), indexes[i]);
110         }
111     }
112 }
```

As a result, the launchpad's owner can deposit valuable NFTs as prizes to convince the launchpad's user to pay a fee for the lucky draw. After that, the launchpad's owner can simply withdraw the valuable NFT and

take the fee.

Furthermore, for the `convertToken()` function, there are no restrictions to prevent the owner from changing the `_convertTokenAddress` state variable during the convert phase, resulting in the user not being aware that the token address has been changed while executing the `convertToken()` function to convert their mystery token to another revealed one.

5.7.2. Remediation

Inspex suggests applying a condition to verify the phase of the launchpad.

Hence, if the current phase is in the convert-time duration, the contract will not allow the launchpad's owner to withdraw the convert NFTs.

However, to give fairness to both the launchpad's owner and the launchpad's users:

- Applying a time range for the `convertToken()` function
- Disallowing the changing of `_convertTokenAddress` state through the `setConvertTokenInfo()` function if the NFT is already pushed to the contract

For example,

- Adding the `onlyConvertTime()` modifier and modify the `setConvertTokenInfo()` function
- `_convertTime` must greater than the `redeemEndTimestamp`
- `_convertEndime` must greater than the `_convertTime`

MintableLaunchpad.sol

```
16  uint256 private _convertTime;
17  uint256 private _convertEndTime;
18
19  modifier onlyConvertTime() {
20      require(block.timestamp >= _convertTime, "Mintable Launchpad: Convert Time
    has not been started yet");
21      require(block.timestamp < _convertEndTime, "Mintable Launchpad: Convert
    Time has been ended");
22      _;
23  }
```

MintableLaunchpad.sol

```
58  function setConvertTokenInfo(address convertTokenAddress, uint256 convertTime,
    uint256 convertEndTime) external onlyOwner {
59      if (_convertTime != 0) {
60          require(block.timestamp < _launchpadData.redeemStartTimestamp,
    "Mintable Launchpad: Mintable Launchpad: redeem time has been started");
61      }
62      if (convertTokenAddress != address(_convertTokenAddress)) {
```

```
63          require(_tokenIds.length == 0, "Mintable Launchpad: convert token's
    address should not be changed after pushing the token");
64      }
65      require(convertTime > _launchpadData.redeemEndTimestamp, "Mintable
    Launchpad: convert time must be after redeem time");
66      require(convertEndTime > convertTime, "Mintable Launchpad: convertEndtime
    must greater than convertTime");
67
68      _convertTokenAddress = IERC721(convertTokenAddress);
69      _convertTime = convertTime;
70      _convertEndTime = convertEndTime
71  }
```

- Apply the `onlyConvertTime()` modifier to the `convertToken()` function

MintableLaunchpad.sol

```
1   function convertToken(uint256 tokenId) onlyConvertTime external {
2       require(address(_convertTokenAddress) != address(0), "Mintable Launchpad:
    Not mistery box");
3       require(_tokenIds.length > 0, "Mintable Launchpad: Not enough token");
4       require(ownerOf(tokenId) == _msgSender(), "Mintable Launchpad: Not an
    owner");
5
6       _burn(tokenId);
7
8       uint256 tokenIndex = _tokenIds.randomIndex(tokenId);
9       uint256 receiveTokenId = _tokenIds[tokenIndex];
10
11      _convertTokenAddress.safeTransferFrom(address(this), _msgSender(),
    receiveTokenId);
12
13      _tokenIds[tokenIndex] = _tokenIds[_tokenIds.length - 1];
14      _tokenIds.pop();
15  }
```

- Identify the time phase

MintableLaunchpad.sol

```
98  function withdrawNft(address tokenAddress, uint256[] memory indexes) external {
99      if (tokenAddress == address(_convertTokenAddress)) {
100         require(block.timestamp < _launchpadData.redeemStartTimestamp ||
    block.timestamp > _convertEndTime, "Mintable Launchpad: Convert time phase has
    not been ended yet");
101         for(uint i = 0; i < indexes.length; i++) {
102             uint256 tokenId = _tokenIds[indexes[i]];
103             _convertTokenAddress.safeTransferFrom(address(this), _msgSender(),
```

```
tokenId);
104             _tokenIds[indexes[i]] = _tokenIds[_tokenIds.length - 1];
105             _tokenIds.pop();
106         }
107     } else {
108         IERC721 token = IERC721(tokenAddress);
109         for(uint i = 0; i < indexes.length; i++) {
110             token.safeTransferFrom(address(this), _msgSender(), indexes[i]);
111         }
112     }
113 }
```

This also applies to the `TransferableLaunchpad` contract.

TransferableLaunchpad.sol

```
40  function withdrawNft(address tokenAddress, uint256[] memory indexes) external {
41      if (tokenAddress == address(_nftAddress)) {
42          require(block.timestamp > _launchpadData.redeemEndTimestamp,
    "Transferable Launchpad: During redeem time");
43          for(uint i = 0; i < indexes.length; i++) {
44              uint256 tokenId = _tokenIds[indexes[i]];
45              _nftAddress.safeTransferFrom(address(this), _msgSender(), tokenId);
46              _tokenIds[indexes[i]] = _tokenIds[_tokenIds.length - 1];
47              _tokenIds.pop();
48          }
49      } else {
50          IERC721 token = IERC721(tokenAddress);
51          for(uint i = 0; i < indexes.length; i++) {
52              token.safeTransferFrom(address(this), _msgSender(), indexes[i]);
53          }
54      }
55  }
```

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

5.8. Insecure Randomness on Redeeming NFTs

ID	IDX-008
Target	MintableLaunchpad TransferableLaunchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-330: Use of Insufficiently Random Values
Risk	**Severity:** Low **Impact:** Medium A whitelisted user can control the random result to select a specific NFT when redeeming the NFTs. This gives an unfair advantage to the platform users. **Likelihood:** Low Only registered users will be able to perform the randomness manipulation, so it is unlikely that the registered users will perform the provided scenario. Furthermore, to get the value from manipulating the randomness, the stored NFT must be revealed beforehand, which is an uncommon strategy for the NFT project.
Status	**Acknowledged** The Coin98 team has acknowledged this issue by using the last claimed block hash as the source of randomness. As a result, the randomness result can be controlled by a list of users, such as miners.

5.8.1. Description

Both `MintableLaunchpad` and `TransferableLaunchpad` contracts use the `randomIndex()` function to randomize the index of the `_tokenIds` array that stores the pushed NFTs by the contract owner.

For the `MintableLaunchpad` contract, it applies the randomness at line 89 with the array length of `_tokenIds` and `tokenId` as the random factor.

MintableLaunchpad.sol

```
81  function convertToken(uint256 tokenId) external {
82      require(address(_convertTokenAddress) != address(0), "Mintable Launchpad:
    Not mistery box");
83      require(_convertTime <= block.timestamp, "Mintable Launchpad: Not convert
    time");
84      require(_tokenIds.length > 0, "Mintable Launchpad: Not enough token");
85      require(ownerOf(tokenId) == _msgSender(), "Mintable Launchpad: Not an
    owner");
86
87      _burn(tokenId);
```

```
88
89      uint256 tokenIndex = _tokenIds.randomIndex(tokenId);
90      uint256 receiveTokenId = _tokenIds[tokenIndex];
91
92      _convertTokenAddress.safeTransferFrom(address(this), _msgSender(),
    receiveTokenId);
93
94      _tokenIds[tokenIndex] = _tokenIds[_tokenIds.length - 1];
95      _tokenIds.pop();
96  }
```

For the `TransferableLaunchpad` contract, it applies the randomness at line 17 with the address of `msg.sender` as the random factor.

TransferableLaunchpad.sol

```
16  function _redeemToken() internal override {
17      uint256 tokenIndex = _tokenIds.randomIndex(uint256(uint160(_msgSender())));
18      uint256 tokenId = _tokenIds[tokenIndex];
19
20      _nftAddress.safeTransferFrom(address(this), _msgSender(), tokenId);
21
22      _tokenIds[tokenIndex] = _tokenIds[_tokenIds.length - 1];
23      _tokenIds.pop();
24  }
```

The `randomIndex()` function takes the parameters as random factors along with the `block.timestamp`.

Randomness.sol

```
6   function randomIndex(uint256[] storage array, uint256 base) internal view
    returns(uint256 result) {
7       uint256 length = array.length;
8       // saving gas
9       assembly {
10          // need allocate memory
11          let emptyPtr := mload(0x40)
12          mstore(0x40, add(emptyPtr, 0xa0))
13
14          mstore(emptyPtr, base)
15          mstore(add(emptyPtr, 0x20), length)
16          mstore(add(emptyPtr, 0x40), timestamp())
17
18          result := mod(keccak256(emptyPtr, 0x60), length)
19      }
20  }
```

Therefore, the random result can be calculated beforehand to find the `tokenId`. This is because all the

random factors are known by the caller. This results in an unfair advantage to the platform's users.

5.8.2. Remediation

Inspex suggests applying the Chainlink VRF as the randomness source (https://docs.chain.link/docs/vrf/v2/introduction/).

However, due to the business design, the owners of the launchpad contract will be end-users who use the `LaunchpadFactory` contract to create the launchpad contracts. This will cause complexity in both the business design and end-user usage.

Therefore, an alternative solution that could mitigate this issue with less complexity is using the block hash of the future block as a source of randomness (in the following source code, the `blockhash()` function is support only to the range of previous 256 blocks at the execution block, so please make a decision about using the condition at line 97 of the `MintableLaunchpad` contract and line 29 of the `TransferableLaunchpad` contract that will block the user from claiming their NFT after the 256 blocks have passed, forever).

For the `MintableLaunchpad` contract:

The launchpad's user executes the `commitConvertToken()` function to set the `_tokenIdSettleBlockNumber` of the desired token to `block.number + 10`, then executes the `convertToken()` function when the current `block.number` is greater than the `_tokenIdSettleBlockNumber` to convert the Mystery Box NFT to the new one.

MintableLaunchpad.sol

```
82  mapping(uint256 => uint256) _tokenIdSettleBlockNumber;
83
84  function commitConvertToken(uint256 tokenId) external {
85      require(ownerOf(tokenId) == _msgSender(), "Mintable Launchpad: Not an
    owner");
86      // Allow to set settle block number to set only once per token, please
    consider about this condition.
87      require(_tokenIdSettleBlockNumber[tokenId] == 0, "Mintable Launchpad: claim
    is already set");
88      //set _tokenIdSettleBlockNumber to the future block
89      _tokenIdSettleBlockNumber[tokenId] = block.number + 10;
90  }
91
92  function convertToken(uint256 tokenId) external {
93      // The user should wait for 10 blocks to settle the random NFT.
94      require(block.number > _tokenIdSettleBlockNumber[tokenId], "Mintable
    Launchpad: settle time is not arrived yet");
95      // If the current block has pass from _tokenIdSettleBlockNumber more than
    256 block, user will not be able to settle again, please consider about this
    condition.
```

```
96      bytes32 settleBlockHash =
        bytes32(blockhash(_tokenIdSettleBlockNumber[tokenId]));
97      require(settleBlockHash != bytes32(0), "Mintable Launchpad:
        _tokenIdSettleBlockNumber was expired");
98
99      require(address(_convertTokenAddress) != address(0), "Mintable Launchpad:
        Not mistery box");
100     require(_convertTime <= block.timestamp, "Mintable Launchpad: Not convert
        time");
101     require(_tokenIds.length > 0, "Mintable Launchpad: Not enough token");
102     require(ownerOf(tokenId) == _msgSender(), "Mintable Launchpad: Not an
        owner");
103
104     _burn(tokenId);
105
106     uint256 base = uint256(keccak256(abi.encodePacked(tokenId,
        settleBlockHash)));
107     uint256 tokenIndex = base % _tokenIds.length;
108     uint256 receiveTokenId = _tokenIds[tokenIndex];
109
110     _convertTokenAddress.safeTransferFrom(address(this), _msgSender(),
        receiveTokenId);
111
112     _tokenIds[tokenIndex] = _tokenIds[_tokenIds.length - 1];
113     _tokenIds.pop();
114 }
```

For the `TransferableLaunchpad` contract:

The launchpad's user executes the `commitRedeemToken()` function to set the `_userSettleBlockNumber` to `block.timestamp + 10`, then executes the `redeem()` function when the current `block.number` greater than the `_userSettleBlockNumber` to redeem NFTs.

TransferableLaunchpad.sol

```
16  mapping(address => uint256) _userSettleBlockNumber;
17
18  function commitRedeemToken() external onlyRegister {
19      require(_userSettleBlockNumber[_msgSender()] == 0, "Transferable Launchpad:
        settle time is already set");
20      //set _userSettleBlockNumber to the future block
21      _userSettleBlockNumber[_msgSender()] = block.number + 10;
22  }
23
24  function _redeemToken() internal override {
25      // The user should wait for 10 blocks to settle the random NFT.
26      require(block.number > _userSettleBlockNumber[_msgSender()], "Transferable
```

```
Launchpad: settle time is not arrived yet");
27      // If the current block has pass from _userSettleBlockNumber more than 256
block, user will not be able to settle again, please consider about this
condition.
28      bytes32 settleBlockHash =
bytes32(blockhash(_userSettleBlockNumber[_msgSender()]));
29      require(settleBlockHash != bytes32(0), "Transferable Launchpad:
_userSettleBlockNumber was expired");
30
31      uint256 base = uint256(keccak256(abi.encodePacked(settleBlockHash)));
32      uint256 tokenIndex = base % _tokenIds.length;
33      uint256 tokenId = _tokenIds[tokenIndex];
34
35      _nftAddress.safeTransferFrom(address(this), _msgSender(), tokenId);
36
37      _tokenIds[tokenIndex] = _tokenIds[_tokenIds.length - 1];
38      _tokenIds.pop();
39  }
```

5.9. Improper Fee Enforcement

ID	IDX-009
Target	Launchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The launchpad's owner will either lose all profits or will have to pay an additional fee to the platform. **Likelihood:** Low It is very unlikely that either the launchpad's owner or the platform will enforce an unreasonable fee that affects both the launchpad's profitability and the platform's reputation.
Status	Resolved The Coin98 team has resolved this issue by adding the boundary to ensure that the protocol fee is less than the launchpad fee.

5.9.1. Description

In the redemption phase of the launchpad project, the registered users can execute the `redeem()` function to redeem their NFTs by paying an extra fee for the platform.

Launchpad.sol

```
146 function redeem(uint256 amount) onlyRedeemTime onlyRegister onlyActiveLaunchpad
    external payable {
147     require(_launchpadData.maxPerUser == 0 || _totalNftRedeemed[_msgSender()] +
    amount <= _launchpadData.maxPerUser, "Launchpad: Over max nft per user");
148     require(_launchpadData.maxRedeem == 0 || _totalRedeem + amount <=
    _launchpadData.maxRedeem, "Launchpad: Reach max redeem");
149     _claimFee(amount);
150
151     for (uint i = 0; i < amount; i++) {
152         _redeemToken();
153     }
154
155     _totalRedeem = _totalRedeem + uint32(amount);
156
157     _totalNftRedeemed[_msgSender()] = _totalNftRedeemed[_msgSender()] + amount;
158
159     emit Redeem(_msgSender(), amount);
```

```
160 }
```

In the `_claimFee()` function, the extra fee will be calculated and paid. The extra fee will be divided into two parts: the `feeRedeem` which is the fee that the users pay for redeeming NFTs; and the `protocolFee` which is a smaller portion that is sliced from the `feeRedeem` to pay back to the platform.

Launchpad.sol

```
72  function _claimFee(uint256 total) internal {
73      uint256 feeRedeem = _launchpadData.feeRedeem * total;
74      if (feeRedeem > 0) {
75          address feeAddress = _factory.getFeeAddress();
76          uint256 protocolFee = (feeRedeem * _protocolFee / 10000) + _sharingFee;
77          if (_launchpadData.feeRedeemAddress == address(0)) {
78              require(msg.value >= feeRedeem, "Launchpad: Exceed fee");
79              (bool sent,) = payable(feeAddress).call{value: protocolFee}("");
80              require(sent, "Launchpad: Fail to send protocol fee");
81          } else {
82              IERC20 token = IERC20(_launchpadData.feeRedeemAddress);
83              token.safeTransferFrom(_msgSender(), address(this), feeRedeem);
84              token.safeTransfer(feeAddress, protocolFee);
85          }
86      }
87  }
```

There are some edge cases that will break the business logic.

For example, if the `_protocolFee` state variable has been set to 10000 or greater, which means the `protocolFee` portion will be greater than or equal to 100% of the `feeRedeem` portion, and the current implementation requires the user to pay the extra fee only for the `feeRedeem` portion, the launchpad will either lose all profits or have to pay an additional portion to the platform if the `protocolFee` value exceeds the `feeRedeem` portion.

Moreover, in the current implementation, there is no boundary for the `_protocolFee` and the `_sharingFee` values in the `setFee()` function, so the owner or the platform can set them to any value.

Launchpad.sol

```
110 function setFee(uint256 sharingFee, uint256 protocolFee) external override
    onlyOwner {
111     _sharingFee = sharingFee;
112     _protocolFee = protocolFee;
113
114     emit SetFee(sharingFee, protocolFee);
115 }
```

5.9.2. Remediation

Inspex suggests insisting the user pay the extra fee as the most valuable of the `feeRedeem` and the `protocolFee` to prevent the launchpad from losing its profits.

Launchpad.sol

```
72  function _claimFee(uint256 total) internal {
73      uint256 feeRedeem = _launchpadData.feeRedeem * total;
74      if (feeRedeem > 0) {
75          address feeAddress = _factory.getFeeAddress();
76          uint256 protocolFee = (feeRedeem * _protocolFee / 10000) + _sharingFee;
77
78          uint256 extraFee = feeRedeem;
79          if (protocolFee > feeRedeem) { extraFee = protocolFee; }
80
81          if (_launchpadData.feeRedeemAddress == address(0)) {
82              require(msg.value >= extraFee, "Launchpad: Exceed fee");
83              (bool sent,) = payable(feeAddress).call{value: protocolFee}("");
84              require(sent, "Launchpad: Fail to send protocol fee");
85          } else {
86              IERC20 token = IERC20(_launchpadData.feeRedeemAddress);
87              token.safeTransferFrom(_msgSender(), address(this), extraFee);
88              token.safeTransfer(feeAddress, protocolFee);
89          }
90      }
91  }
```

For the `setFee()` function, Inspex suggests implementing a boundary for both `_protocolFee` and `_sharingFee` state variables to guarantee the possible value of the state variables will always be within the acceptable range.

For example:

Launchpad.sol

```
110 function setFee(uint256 sharingFee, uint256 protocolFee) external override
    onlyOwner {
111     require(sharingFee <= 100000, "Launchpad: new sharingFee should <=
    100000"); // example value
112     require(protocolFee <= 1500, "Launchpad: new protocolFee should <= 1500");
    // example value
113     _sharingFee = sharingFee;
114     _protocolFee = protocolFee;
115
116     emit SetFee(sharingFee, protocolFee);
117 }
```

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

5.10. Insufficient Parameter Validation in withdrawNft() Function

ID	IDX-010
Target	MintableLaunchpad TransferableLaunchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Low **Impact:** Medium When the `withdrawNft()` function is executed with an `index` that includes the value which is equal to at least `_tokenIds.length - indexes.length` and the order is not in descending order, the transaction will be reverted. In another case, the platform's user withdraws with an `index` in ascending order that does not reach to the `_tokenIds.length`, the selected index will mismatch to the actual one due to the `_tokenIds.pop()`. **Likelihood:** Low It is unlikely that the launchpad's owner who executes this function will pass the `indexes` parameter with incorrect sequence order.
Status	**Acknowledged** The Coin98 team has acknowledged this issue for gas savings purposes.

5.10.1. Description

In both `MintableLaunchpad` and `TransferableLaunchpad` contracts, it allows users to withdraw the NFTs from the contracts.

In order to withdraw, the users are required to specify the NFT address collection with a list of token IDs for that NFT address collection, as shown in the source code below.

MintableLaunchpad.sol

```
 98 function withdrawNft(address tokenAddress, uint256[] memory indexes) external {
 99     if (tokenAddress == address(_convertTokenAddress)) {
100         for(uint i = 0; i < indexes.length; i++) {
101             uint256 tokenId = _tokenIds[indexes[i]];
102             _convertTokenAddress.safeTransferFrom(address(this), _msgSender(),
    tokenId);
103             _tokenIds[indexes[i]] = _tokenIds[_tokenIds.length - 1];
104             _tokenIds.pop();
105         }
106     } else {
```

```
107         IERC721 token = IERC721(tokenAddress);
108         for(uint i = 0; i < indexes.length; i++) {
109             token.safeTransferFrom(address(this), _msgSender(), indexes[i]);
110         }
111     }
112 }
```

TransferableLaunchpad.sol

```
40  function withdrawNft(address tokenAddress, uint256[] memory indexes) external {
41      if (tokenAddress == address(_nftAddress)) {
42          for(uint i = 0; i < indexes.length; i++) {
43              uint256 tokenId = _tokenIds[indexes[i]];
44              _nftAddress.safeTransferFrom(address(this), _msgSender(), tokenId);
45              _tokenIds[indexes[i]] = _tokenIds[_tokenIds.length - 1];
46              _tokenIds.pop();
47          }
48      } else {
49          IERC721 token = IERC721(tokenAddress);
50          for(uint i = 0; i < indexes.length; i++) {
51              token.safeTransferFrom(address(this), _msgSender(), indexes[i]);
52          }
53      }
54  }
```

If the NFT address matches the specified NFT address state, which is `_convertTokenAddress` and `_nftAddress` for the `MintableLaunchpad` and `TransferableLaunchpad` contracts respectively, the contracts will transfer the stored NFT collection with the ids based on `indexes` parameter to the `_msgSender()`.

However, the transfer process is looping with the values in `indexes`. This means if the `indexes` contains the element the value is at least in the range of `_tokenIds.length - indexes.length` and the sequence of indexes is not in descending order, the transaction will be reverted due to the `_tokenIds.pop()` which shortens the length of the array.

In another case, the platform's user withdraw with `index` in ascending order that does not reach to the `_tokenIds.length`, the selected index will mismatch to the actual one due to the `_tokenIds.pop()` characteristic.

5.10.2. Remediation

Inspex suggests applying a sanity check at line 102 in `MintableLaunchpad` contract and line 44 in `TransferableLaunchpad` contract to verify the sequence order of `indexes` parameter. For example,

MintableLaunchpad.sol

```
98  function withdrawNft(address tokenAddress, uint256[] memory indexes) external {
```

```
99      if (tokenAddress == address(_convertTokenAddress)) {
100         uint256 previousIndex = 0;
101         for(uint i = 0; i < indexes.length; i++) {
102             require(previousIndex > indexes[i] || i == 0, "Mintable Launchpad:
    the indexes array is not descending ordered");
103             previousIndex = indexes[i];
104
105             uint256 tokenId = _tokenIds[indexes[i]];
106             _convertTokenAddress.safeTransferFrom(address(this), _msgSender(),
    tokenId);
107             _tokenIds[indexes[i]] = _tokenIds[_tokenIds.length - 1];
108             _tokenIds.pop();
109         }
110     } else {
111         IERC721 token = IERC721(tokenAddress);
112         for(uint i = 0; i < indexes.length; i++) {
113             token.safeTransferFrom(address(this), _msgSender(), indexes[i]);
114         }
115     }
116 }
```

TransferableLaunchpad.sol

```
40  function withdrawNft(address tokenAddress, uint256[] memory indexes) external {
41      if (tokenAddress == address(_nftAddress)) {
42          uint256 previousIndex = 0;
43          for(uint i = 0; i < indexes.length; i++) {
44              require(previousIndex > indexes[i] || i == 0, "Mintable Launchpad:
    the indexes array is not descending ordered");
45              previousIndex = indexes[i];
46
47              uint256 tokenId = _tokenIds[indexes[i]];
48              _nftAddress.safeTransferFrom(address(this), _msgSender(), tokenId);
49              _tokenIds[indexes[i]] = _tokenIds[_tokenIds.length - 1];
50              _tokenIds.pop();
51          }
52      } else {
53          IERC721 token = IERC721(tokenAddress);
54          for(uint i = 0; i < indexes.length; i++) {
55              token.safeTransferFrom(address(this), _msgSender(), indexes[i]);
56          }
57      }
58  }
```

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

5.11. Smart Contract with Unpublished Source Code

ID	IDX-011
Target	AccessControl LaunchpadFactory MintableLaunchpad TimeLock TransferableLaunchpad
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 Coin98 team has acknowledged this issue and decided not to publish the source code because the team wants to protect their intellectual property.

5.11.1. 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.

5.11.2. 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.

5.12. Outdated Compiler Version

ID	IDX-012
Target	AccessControl LaunchpadFactory MintableLaunchpad TimeLock TransferableLaunchpad
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Very Low **Impact:** Low From the list of known Solidity bugs, direct impact cannot be caused from those bugs themselves. **Likelihood:** Low From the list of known Solidity bugs, it is very unlikely that those bugs would affect these smart contracts.
Status	**Acknowledged** The Coin98 team has acknowledged this issue since the inherent bugs in this version will not affect the contracts.

5.12.1. Description

The solidity compiler versions declared in the smart contracts were outdated. These versions have publicly known inherent bugs (https://docs.soliditylang.org/en/latest/bugs.html) that may potentially be used to cause damage to the smart contracts or the users of the smart contracts.

LaunchpadFactory.sol

```
1 // SPDX-License-Identifier: Apache-2.0
2 pragma solidity 0.8.9;
```

The following table contains all targets which the outdated compiler version is declared.

Contract	Version
AccessControl	0.8.9
LaunchpadFactory	0.8.9
MintableLaunchpad	0.8.9

TimeLock	0.8.9
TransferableLaunchpad	0.8.9

5.12.2. Remediation

Inspex suggests upgrading the solidity compiler to the latest stable version (https://github.com/ethereum/solidity/releases). At the time of audit, the latest stable versions of Solidity compiler in major 0.8 is 0.8.17.

5.13. Insufficient Logging for Privileged Functions

ID	IDX-013
Target	Launchpad LaunchpadFactory MintableLaunchpad TransferableLaunchpad
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved The Coin98 team resolved this issue by emitting events for the execution of privileged functions.

5.13.1. 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 of the `MintableLaunchpad` contract can set tokens that users will receive by executing the `pushToken()` function, but no events are emitted, resulting in users not noticing and losing their chance to claim the token as the very first runners.

MintableLaunchpad.sol

```
68 function pushToken(uint256[] memory tokenIds) external onlyOwner {
69     require(address(_convertTokenAddress) != address(0), "Mintable Launchpad:
   Invalid convert token");
70
71     IERC721 token = IERC721(_convertTokenAddress);
72     for(uint256 i; i< tokenIds.length; i++) {
73         token.safeTransferFrom(_msgSender(), address(this), tokenIds[i]);
74         _tokenIds.push(tokenIds[i]);
75     }
76 }
```

The privileged functions without sufficient logging are as follows:

File	Contract	Function
Launchpad.sol (L:207)	Launchpad	withdrawFungibleToken()
LaunchpadFactory.sol (L:147)	LaunchpadFactory	withdrawFungibleToken()
MintableLaunchpad.sol (L:58)	MintableLaunchpad	setConvertTokenInfo()
MintableLaunchpad.sol (L:68)	MintableLaunchpad	pushToken()
TransferableLaunchpad.sol (L:32)	TransferableLaunchpad	pushToken()

5.13.2. Remediation

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

MintableLaunchpad.sol

```
67  event PushToken(address _convertTokenAddress, uint256[] tokenIds);
68
69  function pushToken(uint256[] memory tokenIds) external onlyOwner {
        require(address(_convertTokenAddress) != address(0), "Mintable Launchpad:
70  Invalid convert token");
71
72      IERC721 token = IERC721(_convertTokenAddress);
73      for(uint256 i; i< tokenIds.length; i++) {
74          token.safeTransferFrom(_msgSender(), address(this), tokenIds[i]);
75          _tokenIds.push(tokenIds[i]);
76      }
77      emit PushToken(address(_convertTokenAddress), tokenIds);
    }
```

5.14. Unchecked Return Value ERC20 Transfer

ID	IDX-014
Target	Launchpad LaunchpadFactory
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The Coin98 team has resolved this issue as suggested by replacing the `transfer()` function with `safeTransfer()` function from OpenZeppelin's SafeERC20 library.

5.14.1. Description

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

In both `Launchpad` and `LaunchpadFactory` contracts, the `withdrawFungibleToken()` function can be used to transfer ERC20 tokens in the contract to the owner's address.

Launchpad.sol

```
207 function withdrawFungibleToken(address tokenAddress, uint256 amount) external
    onlyOwner {
208     if (tokenAddress == address(0)) {
209         (bool sent,) = _msgSender().call{value: amount}("");
210         require(sent, "Launchpad: Fail to send ETH");
211     } else {
212         IERC20 token = IERC20(tokenAddress);
213         token.transfer(_msgSender(), amount);
214     }
215 }
```

LaunchpadFactory.sol

```
147 function withdrawFungibleToken(address tokenAddress, uint256 amount) external
    onlyOwner {
148     if (tokenAddress == address(0)) {
149         (bool sent,) = _msgSender().call{value: amount}("");
150         require(sent, "Launchpad: Fail to send ETH");
```

```
151     } else {
152         IERC20 token = IERC20(tokenAddress);
153         token.transfer(_msgSender(), amount);
154     }
155 }
```

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

However, there's no impact in this case since this function is only used for collecting fee tokens or the tokens mistakenly transferred to the contract, and thus is not related to the users' balances.

5.14.2. Remediation

Inspex suggests replacing the `transfer()` function with `safeTransfer()` function from OpenZeppelin's SafeERC20 library in both `Launchpad` and `LaunchpadFactory` contracts, for example:

Launchpad.sol

```
207 function withdrawFungibleToken(address tokenAddress, uint256 amount) external
    onlyOwner {
208     if (tokenAddress == address(0)) {
209         (bool sent,) = _msgSender().call{value: amount}("");
210         require(sent, "Launchpad: Fail to send ETH");
211     } else {
212         IERC20 token = IERC20(tokenAddress);
213         token.safeTransfer(_msgSender(), amount);
214     }
215 }
```

LaunchpadFactory.sol

```
147 function withdrawFungibleToken(address tokenAddress, uint256 amount) external
    onlyOwner {
148     if (tokenAddress == address(0)) {
149         (bool sent,) = _msgSender().call{value: amount}("");
150         require(sent, "Launchpad: Fail to send ETH");
151     } else {
152         IERC20 token = IERC20(tokenAddress);
153         token.safeTransfer(_msgSender(), amount);
154     }
155 }
```

5.15. Insufficient Parameter Validation in redeem() Function

ID	IDX-015
Target	TransferableLaunchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact The Coin98 team has acknowledged this issue since the `amount` parameter can be adjusted from the front end.

5.15.1. Description

In the `TransferableLaunchpad` contract, the platform's user can redeem the NFT through the `redeem()` function.

Launchpad.sol

```
146 function redeem(uint256 amount) onlyRedeemTime onlyRegister onlyActiveLaunchpad
    external payable {
147     require(_launchpadData.maxPerUser == 0 || _totalNftRedeemed[_msgSender()] +
    amount <= _launchpadData.maxPerUser, "Launchpad: Over max nft per user");
148     require(_launchpadData.maxRedeem == 0 || _totalRedeem + amount <=
    _launchpadData.maxRedeem, "Launchpad: Reach max redeem");
149     _claimFee(amount);
150
151     for (uint i = 0; i < amount; i++) {
152         _redeemToken();
153     }
154
155     _totalRedeem = _totalRedeem + uint32(amount);
156
157     _totalNftRedeemed[_msgSender()] = _totalNftRedeemed[_msgSender()] + amount;
158
159     emit Redeem(_msgSender(), amount);
160 }
```

The internal `_redeemToken()` function will be called to transfer the NFT from the contract to the target address with the `amount` passed from the `redeem()` function.

TransferableLaunchpad.sol

```
1  function _redeemToken() internal override {
2      uint256 tokenIndex = _tokenIds.randomIndex(uint256(uint160(_msgSender())));
3      uint256 tokenId = _tokenIds[tokenIndex];
4
5      _nftAddress.safeTransferFrom(address(this), _msgSender(), tokenId);
6
7      _tokenIds[tokenIndex] = _tokenIds[_tokenIds.length - 1];
8      _tokenIds.pop();
9  }
```

The NFT that is sent to the user will come from the platform's owner through the `pushToken()` function.

TransferableLaunchpad.sol

```
32 function pushToken(uint256[] memory tokenIds) external onlyOwner {
33     IERC721 token = IERC721(_nftAddress);
34     for(uint256 i; i< tokenIds.length; i++) {
35         token.safeTransferFrom(_msgSender(), address(this), tokenIds[i]);
36         _tokenIds.push(tokenIds[i]);
37     }
38 }
```

This means if the platform's user redeems the NFT with the amount that is insufficient, the transaction will always fail.

5.15.2. Remediation

Inspex suggests modifying the `amount` value when it exceeds the current stored NFT in the contract to the total NFT balance.

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