Code Sekai

NFT Minting & Transferring In-game/Out-game

Smart Contract Audit Report

Date Issued: 17 Apr 2023

Version: Final v1.0

Public

ValiX Consulting

Table of Contents

Executive Summary **3**
Overview 3
About NFT Minting & Transferring In-game/Out-game 3
Scope of Work 3
Auditors 5
Disclaimer 5
Audit Result Summary 6
Methodology **7**
Audit Items 8
Risk Rating 10
Findings **11**
Review Findings Summary 11
Detailed Result 13
Appendix **104**
About Us 104
Contact Information 104
References 105

Executive Summary

Overview

Valix conducted a smart contract audit to evaluate potential security issues of the **NFT Minting & Transferring In-game/Out-game features**. This audit report was published on *17 Apr 2023*. The audit scope is limited to the **NFT Minting & Transferring In-game/Out-game features.** Our security best practices strongly recommend that the **Code Sekai team** conduct a full security audit for both on-chain and off-chain components of its infrastructure and their interaction. A comprehensive examination has been performed during the audit process utilizing Valix's Formal Verification, Static Analysis, and Manual Review techniques.

About NFT Minting & Transferring In-game/Out-game

NFT Minting: This allows users to mint their NFT with their on-chain randomized metadata and there are 3 rounds of minting for different groups of users which are Whitelist, Waitlist, and Public.

Transferring: This allows users to transfer their NFT into the Codesekai game and while NFT is in-game they can't transfer NFT to any wallets, if the users want to transfer NFT they must change the status back to out-game.

Scope of Work

The security audit conducted does not replace the full security audit of the overall Code Sekai protocol. The scope is limited to the **NFT Minting & Transferring In-game/Out-game features** and their related smart contracts.

The security audit covered the components at this specific state:

Item	Description
Components	▪ *NFT Minting & Transferring In-game/Out-game smart contracts* ▪ *Imported associated smart contracts and libraries*
Git Repository	▪ *http://gitlab.echoplus.io/echoplus/backend/codesekai-smart-contract-audit.git*

Audit Commit	▪ *a65adff9df58137e67928fa48dc8987f60632e90 (branch: audit)*
Reassessment Commit	▪ *d2ee90e6974cc89e8df246f60e47bc8eeee38471 (branch: audit)*
Audited Files	▪ *./contracts/CSKGen.sol* ▪ *./contracts/CSKNFT.sol* ▪ *Other imported associated Solidity files*
Excluded Files/Contracts	▪ -

Remark: Our security best practices strongly recommend that the Code Sekai team conduct a full security audit for both on-chain and off-chain components of its infrastructure and the interaction between them.

Auditors

Role	Staff List
Auditors	**Parichaya Thanawuthikrai (Lead Auditor)** **Anak Mirasing** **Kritsada Dechawattana**
Authors	**Anak Mirasing** **Kritsada Dechawattana** **Parichaya Thanawuthikrai**
Reviewers	**Phuwanai Thummavet (Technical Advisor)** **Sumedt Jitpukdebodin**

Disclaimer

Our smart contract audit was conducted over a limited period and was performed on the smart contract at a single point in time. As such, the scope was limited to current known risks during the work period. The review does not indicate that the smart contract and blockchain software has no vulnerability exposure.

We reviewed the security of the smart contracts with our best effort, and we do not guarantee a hundred percent coverage of the underlying risk existing in the ecosystem. The audit was scoped only in the provided code repository. The on-chain code is not in the scope of auditing.

This audit report does not provide any warranty or guarantee, nor should it be considered an “approval” or “endorsement” of any particular project. This audit report should also not be used as investment advice nor provide any legal compliance.

Audit Result Summary

From the audit results and the remediation and response from the developer, Valix trusts that the **NFT Minting & Transferring In-game/Out-game features** have sufficient security protections to be safe for use.

Initially, Valix was able to identify **32 issues** that were categorized from the "Critical" to "Informational" risk level in the given timeframe of the assessment.

For the reassessment, the *CodeSekai* team acknowledged a total of 32 issues, including 1 critical issue, 9 high issues, 7 medium issues, 9 low issues, and 6 informational issues. Of these, the team was able to completely fix 27 issues, partially fix 3 issues, and acknowledge 2 issues.

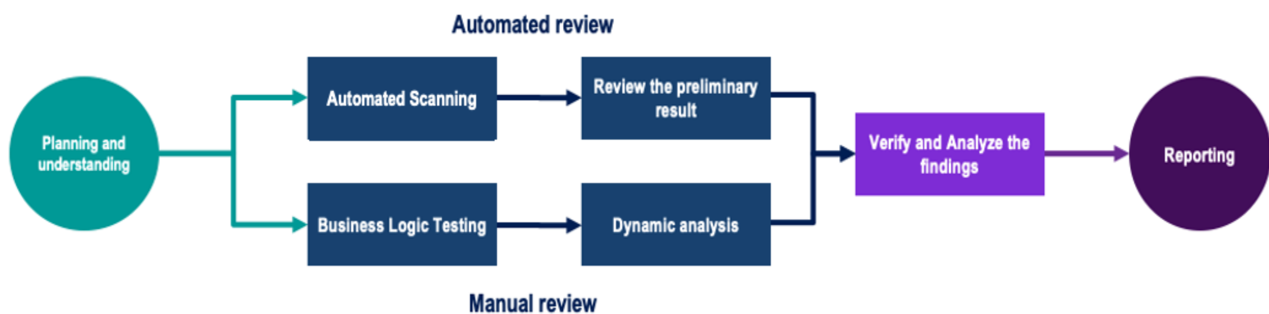
Below is the breakdown of the vulnerabilities found and their associated risk rating for each assessment conducted.

Target	Assessment Result					Reassessment Result				
	C	H	M	L	I	C	H	M	L	I
NFT Minting & Transferring In-game/Out-game	1	9	7	9	6	0	2	0	1	2

Note: *Risk Rating* ***C*** *Critical,* ***H*** *High,* ***M*** *Medium,* ***L*** *Low,* ***I*** *Informational*

Methodology

The smart contract security audit methodology is based on Smart Contract Weakness Classification and Test Cases (SWC Registry), CWE, well-known best practices, and smart contract hacking case studies. Manual and automated review approaches can be mixed and matched, including business logic analysis in terms of the malicious doer's perspective. Using automated scanning tools to navigate or find offending software patterns in the codebase along with a purely manual or semi-automated approach, where the analyst primarily relies on one's knowledge, is performed to eliminate the false-positive results.



Planning and Understanding

- Determine the scope of testing and understanding of the application's purposes and workflows.
- Identify key risk areas, including technical and business risks.
- Determine which sections to review within the resource constraints and review method – automated, manual or mixed.

Automated Review

- Adjust automated source code review tools to inspect the code for known unsafe coding patterns.
- Verify the tool's output to eliminate false-positive results, and adjust and re-run the code review tool if necessary.

Manual Review

- Analyzing the business logic flaws requires thinking in unconventional methods.
- Identify unsafe coding behavior via static code analysis.

Reporting

- Analyze the root cause of the flaws.
- Recommend improvements for secure source code.

Audit Items

We perform the audit according to the following categories and test names.

Category	ID	Test Name
Security Issue	SEC01	*Authorization Through tx.origin*
	SEC02	*Business Logic Flaw*
	SEC03	*Delegatecall to Untrusted Callee*
	SEC04	*DoS With Block Gas Limit*
	SEC05	*DoS with Failed Call*
	SEC06	*Function Default Visibility*
	SEC07	*Hash Collisions With Multiple Variable Length Arguments*
	SEC08	*Incorrect Constructor Name*
	SEC09	*Improper Access Control or Authorization*
	SEC10	*Improper Emergency Response Mechanism*
	SEC11	*Insufficient Validation of Address Length*
	SEC12	*Integer Overflow and Underflow*
	SEC13	*Outdated Compiler Version*
	SEC14	*Outdated Library Version*
	SEC15	*Private Data On-Chain*
	SEC16	*Reentrancy*
	SEC17	*Transaction Order Dependence*
	SEC18	*Unchecked Call Return Value*
	SEC19	*Unexpected Token Balance*
	SEC20	*Unprotected Assignment of Ownership*
	SEC21	*Unprotected SELFDESTRUCT Instruction*
	SEC22	*Unprotected Token Withdrawal*
	SEC23	*Unsafe Type Inference*
	SEC24	*Use of Deprecated Solidity Functions*
	SEC25	*Use of Untrusted Code or Libraries*
	SEC26	*Weak Sources of Randomness from Chain Attributes*
	SEC27	*Write to Arbitrary Storage Location*

Category	ID	Test Name
Functional Issue	FNC01	*Arithmetic Precision*
	FNC02	*Permanently Locked Fund*
	FNC03	*Redundant Fallback Function*
	FNC04	*Timestamp Dependence*
Operational Issue	OPT01	*Code With No Effects*
	OPT02	*Message Call with Hardcoded Gas Amount*
	OPT03	*The Implementation Contract Flow or Value and the Document is Mismatched*
	OPT04	*The Usage of Excessive Byte Array*
	OPT05	*Unenforced Timelock on An Upgradeable Proxy Contract*
Developmental Issue	DEV01	*Assert Violation*
	DEV02	*Other Compilation Warnings*
	DEV03	*Presence of Unused Variables*
	DEV04	*Shadowing State Variables*
	DEV05	*State Variable Default Visibility*
	DEV06	*Typographical Error*
	DEV07	*Uninitialized Storage Pointer*
	DEV08	*Violation of Solidity Coding Convention*
	DEV09	*Violation of Token (ERC20) Standard API*

Risk Rating

To prioritize the vulnerabilities, we have adopted the scheme of five distinct levels of risk: **Critical**, **High**, **Medium**, **Low**, and **Informational**, based on OWASP Risk Rating Methodology. The risk level definitions are presented in the table.

Risk Level	Definition
Critical	The code implementation does not match the specification, and it could disrupt the platform.
High	The code implementation does not match the specification, or it could result in losing funds for contract owners or users.
Medium	The code implementation does not match the specification under certain conditions, or it could affect the security standard by losing access control.
Low	The code implementation does not follow best practices or use suboptimal design patterns, which may lead to security vulnerabilities further down the line.
Informational	Findings in this category are informational and may be further improved by following best practices and guidelines.

The **risk value** of each issue was calculated from the product of the **impact** and **likelihood values**, as illustrated in a two-dimensional matrix below.

- **Likelihood** represents how likely a particular vulnerability is exposed and exploited in the wild.
- **Impact** measures the technical loss and business damage of a successful attack.
- **Risk** demonstrates the overall criticality of the risk.

Impact \ Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Informational

The shading of the matrix visualizes the different risk levels. Based on the acceptance criteria, the risk levels "Critical" and "High" are unacceptable. Any issue obtaining the above levels must be resolved to lower the risk to an acceptable level.

Findings

Review Findings Summary

The table below shows the summary of our assessments.

No.	Issue	Risk	Status	Functionality is in use
1	Potential Replay Attack On NFT Updating	Critical	Fixed	In use
2	Unprotected Initialization Of Crucial State Variables	High	Fixed	In use
3	Lack Of Setter Function For signWallet State	High	Fixed	In use
4	Permanently Losing The Admin Role	High	Fixed	In use
5	Potential Replay Attack On NFT Minting	High	Fixed	In use
6	Possibly Bypassing The Condition To Generate Token	High	Fixed	In use
7	Non-Uniqueness NFT Metadata Assignment	High	Acknowledged	In use
8	Trust And Fairness Of Metadata Generation	High	Acknowledged	In use
9	Recommended Improving Transparency And Trustworthiness	High	Fixed	In use
10	Burning Tokens Without Validating Availability Flag	High	Fixed	In use
11	Improper Verification Of Supply Checking	Medium	Fixed	In use
12	Inconsistent State In Token Management When Burning Tokens	Medium	Fixed	In use
13	Incorrect Condition For Removing Whitelist	Medium	Fixed	In use
14	No Upper Bound For The Portal Price	Medium	Fixed	In use
15	Lack Of Setter Function For adminWallet State	Medium	Fixed	In use
16	Directly Minting Without Permission	Medium	Fixed	In use
17	Possibly Setting Improper Period For Each Minting Round	Medium	Fixed	In use
18	Overpayment When Minting And Updating NFT	Low	Fixed	In use
19	Lack Of Validating Input Parameters	Low	Fixed	In use
20	Compiler Is Not Locked To Specific Version	Low	Fixed	In use

21	Compiler May Be Susceptible To Publicly Disclosed Bugs	Low	Fixed	In use
22	Arbitrarily Setting NFT Minting Prices	Low	Fixed	In use
23	Potential Denial-Of-Service On The getUserTokenAndInfos Function	Low	Fixed	In use
24	Recommended Improving Transparency And Traceability Of Crucial Variables	Low	Partially Fixed	In use
25	Recommended Event Emissions For Transparency And Traceability	Low	Fixed	In use
26	Lack Of Checking Availability Of The Token ID	Low	Fixed	In use
27	Recommended Removing Unused Interfaces	Informational	Fixed	In use
28	Recommended Removing Unused Code	Informational	Fixed	In use
29	Recommended Removing Unused Imported Contract	Informational	Partially Fixed	In use
30	Misspelling Of Crucial Function Name	Informational	Fixed	In use
31	Recommended Adding Event Indexes	Informational	Fixed	In use
32	Recommended Enforcing Checks-Effects-Interactions Pattern	Informational	Partially Fixed	In use

The statuses of the issues are defined as follows:

Fixed: The issue has been completely resolved and has no further complications.

Partially Fixed: The issue has been partially resolved.

Acknowledged: The issue’s risk has been reported and acknowledged.

Detailed Result

This section provides all issues that we found in detail.

No. 1	Potential Replay Attack On NFT Updating		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKNFT.sol*		
Locations	*CSKNFT.sol* ● *L: 76 - 81 (*The *SignInfo* struct*)* ● *L: 156 - 170 (*The *_hash* function*)* ● *L: 172 - 175 (*The *_verify* function*)* ● *L: 277 - 303 (*The *updateFlagStatus* function*)*		

Detailed Issue

We found **the potential replay attack issue affects the *updateFlagStatus* function** (L277 - 303 in the code snippet 1.1).

The *updateFlagStatus* function can be executed by users to update a *CodeSekaiNFT* token availability and metadata and then the *updateFlagStatus* function will verify the signature of the payload, the so-called *SignInfo* (L76 - 81 in the code snippet 1.1). The *SignInfo* is of type struct as follows:

```
struct SignInfo {
    uint256 tokenId;
    string metadata;
    bool status;
    bytes signature;
}
```

To successfully update the status of a token, the sender and the token owner must be the same address (L282), the given status must not be the previous status (L283 - 286), and the payload must be signed by the owner's private key (L290).

However, we discovered **the root cause of this issue is that the payload allows for multiple uses since there is no tracking of the *expiration time* and *nonce* of the payload**. As a result, an **attacker can use the same payload multiple times to update the token**.

CSKNFT.sol

```
76   struct SignInfo {
77       uint256 tokenId;
78       string metadata;
79       bool status;
80       bytes signature;
81   }

     // (...SNIPPED...)

156  function _hash(SignInfo memory info) internal view returns (bytes32) {
157      return
158          _hashTypedDataV4(
159              keccak256(
160                  abi.encode(
161                      keccak256(
163                          "SignInfo(uint256 tokenId,string metadata,bool status)"
163                      ),
164                      info.tokenId,
165                      keccak256(bytes(info.metadata)),
166                      info.status
167                  )
168              )
169          );
170  }
171
172  function _verify(SignInfo memory order) internal view returns (address) {
173      bytes32 digest = _hash(order);
174      return ECDSA.recover(digest, order.signature);
175  }

     // (...SNIPPED...)

277  function updateFlagStatus(SignInfo calldata _info)
278      external
279      payable
280      nonReentrant
281  {
282      require(ownerOf(_info.tokenId) == msg.sender, "Not Owner.");
283      require(
284          tokenInfo[_info.tokenId].isAvailable != _info.status,
285          "Same status."
286      );
287
288      //verify
289      address signer = _verify(_info);
290      require(signer == signWallet, "not signed");
291
292      if (_info.status) {
293          require(msg.value >= PORTAL_PRICE, "Eth not enough.");
```

```
294
295         (bool sent, ) = adminWallet.call{value: msg.value}("");
296         require(sent, "Failed send");
297         tokenInfo[_info.tokenId].metadata = _info.metadata;
298     }
299
300     tokenInfo[_info.tokenId].isAvailable = _info.status;
301
302     emit ChangeItemStatus(msg.sender, _info);
303 }
```

Listing 1.1 The affected *updateFlagStatus* function and its related dependencies

Recommendations

We recommend **adding the *nonce* and *expiration time* (L80 and L81 in the code snippet 1.2) parameters to the *SignInfo* struct. The *nonce* and *expiration time* would prevent an attacker from making the replay attack** since the payload will be for single use and limit the deadline of the payload.

To use this *nonce*, we have to add the *updateTokenNonces* mapping (L93) to track the spending of each signed payload, and the *nonce* must be increased every time the payload is consumed (L297).

Finally, when recovering the signer of the payload, the *nonce* (for each token owner) has to be computed in the *_hash* function (L159 - 175). Moreover, we have to add the verification of the *expiration time* (L292) to verify that the payload is not expired.

CSKNFT.sol

```
76  struct SignInfo {
77      uint256 tokenId;
78      string metadata;
79      bool status;
80      uint256 nonce;
81      uint256 expirationTime;
82      bytes signature;
83  }

    // (...SNIPPED...)

93  mapping(address => uint256) public updateTokenNonces;

    // (...SNIPPED...)

159 function _hash(SignInfo memory info) internal view returns (bytes32) {
160     return
161         _hashTypedDataV4(
163             keccak256(
163                 abi.encode(
```

```
164                     keccak256(
165                         "SignInfo(uint256 tokenId,string metadata,bool status,
    uint256 nonce, uint256 expirationTime)"
166                     ),
167                     info.tokenId,
168                     keccak256(bytes(info.metadata)),
169                     info.status,
170                     info.nonce,
171                     info.expirationTime
172                 )
173             )
174         );
175 }
176
177 function _verify(SignInfo memory order) internal view returns (address) {
178     bytes32 digest = _hash(order);
179     return ECDSA.recover(digest, order.signature);
180 }

    // (...SNIPPED...)

282 function updateFlagStatus(SignInfo calldata _info)
283     external
284     payable
285     nonReentrant
286 {
287     require(ownerOf(_info.tokenId) == msg.sender, "Not Owner.");
288     require(
289         tokenInfo[_info.tokenId].isAvailable != _info.status,
290         "Same status."
291     );
292     require(block.timestamp < _info.expirationTime, "Times out");
293
294     //verify
295     address signer = _verify(_info);
296     require(signer == signWallet, "not signed");
297     require(_info.nonce == updateTokenNonces[msg.sender]++, "Invalid nonce");
298
299     if (_info.status) {
300         require(msg.value >= PORTAL_PRICE, "Eth not enough.");
301
302         (bool sent, ) = adminWallet.call{value: msg.value}("");
303         require(sent, "Failed send");
304         tokenInfo[_info.tokenId].metadata = _info.metadata;
305     }
306
307     tokenInfo[_info.tokenId].isAvailable = _info.status;
308
309     emit ChangeItemStatus(msg.sender, _info);
310 }
```

Listing 1.2 The improved *updateFlagStatus* function and its related dependencies

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Regarding the configuration of the *expirationTime* parameter, the parameter can be freely set by the signer (off-chain service). However, **we noticed the possible attacks that can be initiated at the off-chain side if the *CodeSekai* team sets the *expirationTime* parameter too large.** Consider the following scenario to understand the issue.

1. Assuming that the *expirationTime* parameter is set to 3600 seconds (1 hour) for each signed payload.
2. An attacker bridges their NFT from the on-chain (smart contract) to the off-chain service.
3. The attacker asks for the signer (off-chain service) to sign their payload for bridging their NFT back to the on-chain service (smart contract).

   In this step, **the signer signs the payload containing the *NFT metadata* and sets the payload's *expirationTime* parameter to 1 hour ahead of the signing time. This way, the attacker would have 1 hour to perform the attack.**
4. The attacker joins and plays the game.
5. **Suppose that the attacker lost their items to the game.**
6. **The attacker adopts the payload previously signed in Step 3 to bridge their status back to the on-chain service (smart contract).**
7. Since **the payload was validly signed by the legitimate signer and its *expirationTime* parameter is not reached**, the transaction is successfully executed.

   Subsequently, **the attacker can maliciously retrieve back their lost items. This attack can lead to several double-spending issues.**

For this reason, **we recommend the *CodeSekai* team to set the value of the *expirationTime* parameter properly. If possible, we recommend the team to apply mitigation solutions to their off-chain services.** For example, making sure that users would no longer execute any off-chain services if they have triggered the off-chain signer.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

However, **we recommend setting the value of the *expirationTime* parameter with a proper value** to mitigate any possible attacks originating from off-chain services. Refer to the *Recommendation* section above for the detailed explanation.

If possible, we recommend the team to apply mitigation solutions to their off-chain services. For example, ensuring users would no longer execute any off-chain services if they have triggered the off-chain signer.

No. 2	Unprotected Initialization Of Crucial State Variables		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKNFT.sol*		
Locations	*CSKNFT.sol L: 48 - 56, L: 277 - 303*		

Detailed Issue

The *CSKNFT* contract contains an *initialize* function that can only be called once to set the values of the *baseTokenURI*, *adminWallet*, and *signWallet* states (L28, 29, 30 in code snippet 2.1). These states are crucial for the proper functioning of the contract, as the *adminWallet* is used to receive funds (L294 in code snippet 2.1), and the *signWallet* is used to verify sign information (L289 in code snippet 2.1).

However, **the visibility of the *initialize* function is set to *public*** (L52 in code snippet 2.1), **which means that anyone can call it, even before the contract owner has a chance to set these values. This vulnerability could allow an attacker to set arbitrary values for these states, which could cause the system to malfunction.**

For example, if an attacker knows the correct *signWallet* address, they could exploit the vulnerability in the *initialize* function to set the *adminWallet* to an address under their control. This would enable them to receive funds when the *updateFlagStatus* function is invoked by a user, as they could bypass the signature verification requirement (L289 in code snippet 2.1).

Additionally, an attacker could set the *baseTokenURI* to point to a malicious website, causing unsuspecting users to download malware or provide sensitive information.

As a result, this could be harmful to the system or cause it to malfunction.

CSKNFT.sol

```
28  string public baseTokenURI;
29  address payable private adminWallet;
30  address private signWallet;

    // (...SNIPPED...)

48  function initialize(
```

```
49      string memory _baseTokenUri,
50      address payable _adminWallet,
51      address payable _signWallet
52  ) public initializer {
53      baseTokenURI = _baseTokenUri;
54      adminWallet = _adminWallet;
55      signWallet = _signWallet;
56  }
```

Listing 2.1 The *initialize* function of *CSKNFT* contract

CSKNFT.sol

```
277 function updateFlagStatus(SignInfo calldata _info)
278     external
279     payable
280     nonReentrant
281 {
282     require(ownerOf(_info.tokenId) == msg.sender, "Not Owner.");
283     require(
284         tokenInfo[_info.tokenId].isAvailable != _info.status,
285         "Same status."
286     );
287
288     //verify
289     address signer = _verify(_info);
290     require(signer == signWallet, "not signed");
291
292     if (_info.status) {
293         require(msg.value >= PORTAL_PRICE, "Eth not enough.");
294
295         (bool sent, ) = adminWallet.call{value: msg.value}("");
296         require(sent, "Failed send");
297         tokenInfo[_info.tokenId].metadata = _info.metadata;
298     }
299
300     tokenInfo[_info.tokenId].isAvailable = _info.status;
301
302     emit ChangeItemStatus(msg.sender, _info);
303 }
```

Listing 2.2 The *updateFlagStatus* function of *CSKNFT* contract

Recommendations

We recommend **removing the *initialize* function and moving the code from the *initialize* function to the *constructor***. This is because the *CSKNFT* contract is not an upgradeable contract, and it's unnecessary to use an *initialization* function.

By moving the code to the *constructor*, we can ensure that the necessary values are set correctly from the outset, and prevent any external calls to initialize that could potentially cause damage to the system.

Additionally, we recommend adding a check to ensure that the *signWallet* address is not set to the zero address. This is important because the *CSKNFT* contract does not have a setter function to modify the *signWallet* address after deployment.

CSKNFT.sol

```
39 constructor(
40     string memory _baseTokenUri,
41     address payable _adminWallet,
42     address _signWallet
43     )
44     ERC721("CodeSekaiNFT", "CSKI")
45     EIP712(SIGNING_DOMAIN, SIGNATURE_VERSION)
46 {
47     require(_signWallet != address(0), "Invalid _signWallet address");
48     _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
49     _grantRole(MINTER_ROLE, msg.sender);
50     _grantRole(DEV_ROLE, msg.sender);
51
52     baseTokenURI = _baseTokenUri;
53     adminWallet = _adminWallet;
54     signWallet = _signWallet;
55 }
```

Listing 2.3 The improved *constructor* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

<table>
<tr><td>No. 3</td><td colspan="3">Lack Of Setter Function For signWallet State</td></tr>
<tr><td rowspan="2">Risk</td><td rowspan="2">High</td><td>Likelihood</td><td>Medium</td></tr>
<tr><td>Impact</td><td>High</td></tr>
<tr><td>Functionality is in use</td><td>In use</td><td>Status</td><td>Fixed</td></tr>
<tr><td>Associated Files</td><td colspan="3">contracts/CSKNFT.sol</td></tr>
<tr><td>Locations</td><td colspan="3">CSKNFT.sol L: 30, 55, and 289 - 290</td></tr>
</table>

Detailed Issue

The *updateFlagStatus* function of the *CSKNFT* contract allows the NFT owner to update an NFT's metadata and status by providing the information, including the signature. Then there would be verifying that the signature must be signed with the *signWallet* address before updating the NFT's information (L289 - 290 in the code snippet below).

However, we noticed that the *signWallet* address (L30 in the code snippet below) is assigned once at the *initialize* function (L55 in the code snippet below) and cannot change later.

Consequently, if the off-chain signer address is changed, the *updateFlagStatus* invoking will be reverted because the off-chain signer and *signWallet* address are not the same address (L289 - 290 in the code snippet below).

CSKNFT.sol

```
15 contract CSKNFT is
16     ERC721,
17     EIP712,
18     ERC721Enumerable,
19     ERC721Burnable,
20     AccessControl,
21     Ownable,
22     Initializable,
23     ReentrancyGuard
24 {
25     using Counters for Counters.Counter;
26
27     /// @dev Base token URI used as a prefix by tokenURI().
28     string public baseTokenURI;
29     address payable private adminWallet;
30     address private signWallet;
```

```
    //(...SNIPPED...)

48  function initialize(
49      string memory _baseTokenUri,
50      address payable _adminWallet,
51      address payable _signWallet
52  ) public initializer {
53      baseTokenURI = _baseTokenUri;
54      adminWallet = _adminWallet;
55      signWallet = _signWallet;
56  }

    //(...SNIPPED...)

277 function updateFlagStatus(SignInfo calldata _info)
278     external
279     payable
280     nonReentrant
281 {
282     require(ownerOf(_info.tokenId) == msg.sender, "Not Owner.");
283     require(
284         tokenInfo[_info.tokenId].isAvailable != _info.status,
285         "Same status."
286     );
287
288     //verify
289     address signer = _verify(_info);
290     require(signer == signWallet, "not signed");
291
292     if (_info.status) {
293         require(msg.value >= PORTAL_PRICE, "Eth not enough.");
294
295         (bool sent, ) = adminWallet.call{value: msg.value}("");
296         require(sent, "Failed send");
297         tokenInfo[_info.tokenId].metadata = _info.metadata;
298     }
299
300     tokenInfo[_info.tokenId].isAvailable = _info.status;
301
302     emit ChangeItemStatus(msg.sender, _info);
303 }
```

Listing 3.1 The mechanism to verify the signer

Recommendations

We recommend adding the setter function to enable changing the *signWallet* address.

Furthermore, we suggest governing the *setSignWallet* function with the *TIMELOCK_DEV_ROLE*. The *TIMELOCK_DEV_ROLE* is assigned as the only role authorized to execute the associated functions. This would improve the transparency and trustworthiness of privileged operations.

For more information about the usage of the *TIMELOCK_DEV_ROLE*, please refer to issue #9 - Recommended Improvements for Transparency and Trustworthiness.

CSKNFT.sol

```
    event ChangeSignWallet(address indexed prevSignWallet, address indexed
    newSignWallet, address indexed executor);

    //(...SNIPPED...)

322 function setSignWallet(address _signWallet)
323     public
324     onlyRole(TIMELOCK_DEV_ROLE)
325 {
326     require(_signWallet != address(0), "Invalid address");
327     address prevSignWallet = signWallet;
328     signWallet = _signWallet;
329
330     emit ChangeSignWallet(prevSignWallet, signWallet, msg.sender);
331 }
```

Listing 3.2 The new *setSignWallet* function allows for changing the *signWallet* address

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 4	Permanently Losing The Admin Role		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*@openzeppelin/contracts/access/AccessControl.sol*		
Locations	*AccessControl.sol L: 179 - 183 (*The *renounceRole* function*)*		

Detailed Issue

The *CSKGen* and *CSKNFT* contracts derive the *renounceRole* function (L179 - 183 in the code snippet 4.1) function from the *AccessControl* contract. This function can be invoked by anyone to remove their specific role.

We consider the *renounceRole* function risky since it can remove privileged roles, including the *DEFAULT_ADMIN_ROLE*, which is the top-level role. Consider the case that the only account with the *DEFAULT_ADMIN_ROLE* role is removed by calling the *renounceRole* function.

The *CSKGen* and *CSKNFT* contracts will be dangled immediately since the contract will have no account with the *DEFAULT_ADMIN_ROLE* role anymore, and this is unrecoverable.

AccessControl.sol

```
179  function renounceRole(bytes32 role, address account) public virtual override {
180      require(account == _msgSender(), "AccessControl: can only renounce roles for
     self");
181
182      _revokeRole(role, account);
183  }

     // (...SNIPPED...)

241  function _revokeRole(bytes32 role, address account) internal virtual {
242      if (hasRole(role, account)) {
243          _roles[role].members[account] = false;
244          emit RoleRevoked(role, account, _msgSender());
245      }
246  }
```

Listing 4.1 The *renounceRole* function of the *AccessControl* contract

Moreover, the *revokeRole* function of the *AccessControl* contract still has the ability to remove the *DEFAULT_ADMIN_ROLE*, despite the issue we discovered. However, It is important to carefully use this function since it could lead to unexpected consequences.

Recommendations

We recommend **overriding and implementing the *renounceRole* function to the *CSKGen* and *CSKNFT* contracts** as the following code snippet to avoid the case that the sole account with the *DEFAULT_ADMIN_ROLE* role is removed accidentally.

CSKGen.sol

```
230 function renounceRole(bytes32 role, address account) public virtual
    override(AccessControl) {
231     require(!(hasRole(DEFAULT_ADMIN_ROLE, account)), "AccessControl: cannot
    renounce the DEFAULT_ADMIN_ROLE account");
232
233     super.renounceRole(role, account);
234 }
```

Listing 4.2 The example overridden *renounceRole* function of the *CSKGen* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Furthermore, since no specific code or solution can completely fix the *revokeRole* function issue without breaking the contract's features, we suggest taking necessary precautions while using the function.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 5	Potential Replay Attack On NFT Minting		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol*		
Locations	*CSKGen.sol* ● *L: 48 - 54 (*The *MintInfo* struct*)* ● *L: 167 - 182 (*The *_hash* function*)* ● *L: 164 - 187 (*The *_verify* function*)* ● *L: 190 - 228 (*The *genToken* function*)*		

Detailed Issue

We found **the potential replay attack issue affects the *genToken* function** (L190 - 228 in the code snippet 5.1).

The *genToken* function can be executed by users to mint a *CodeSekaiNFT* token and then the *genToken* function will verify the signature of the payload, the so-called *MintInfo* (L48 - 54 in the code snippet 5.1). The *MintInfo* is of type struct as follows:

```
struct MintInfo {
    address minter;
    uint256 timestamp;
    uint256 mintType;
    uint256 metadata;
    bytes signature;
}
```

In order for the minting process to be successful, the payload must not be expired (L193), the sender and the token minter must have the same address (L194), and the payload must be signed by the owner's private key (L198).

However, we **consider the scenario that the signer accidentally signs the *MintInfo* Payload with an incorrect *MintType* (i.e., mintType greater than 2).** In this case, even though there is a verification state for each minting round (L201, L209, and L218), **the token will be minted without proper permission and payment, and the minter could potentially use the same *MintInfo* payload to execute multiple replay attacks**.

CSKGen.sol

```
48  struct MintInfo {
49      address minter;
50      uint256 timestamp;
51      uint256 mintType;
52      uint256 metadata;
53      bytes signature;
54  }

    // (...SNIPPED...)

167 function _hash(MintInfo memory info) internal view returns (bytes32) {
168     return
169         _hashTypedDataV4(
170             keccak256(
171                 abi.encode(
172                     keccak256(
173                         "MintInfo(address minter,uint256 timestamp,uint256
    mintType,uint256 metadata)"
174                     ),
175                     info.minter,
176                     info.timestamp,
177                     info.mintType,
178                     info.metadata
179                 )
180             )
181         );
182 }
183
184 function _verify(MintInfo memory info) internal view returns (address) {
185     bytes32 digest = _hash(info);
186     return ECDSA.recover(digest, info.signature);
187 }
188
189
190 function genToken(MintInfo calldata info) external payable nonReentrant {
191     uint256 ethAmount;
192     MintType mintType;
193     require(block.timestamp <= info.timestamp + 1 minutes, "Times out");
194     require(info.minter == msg.sender, "not minter");
195
196     //verify
197     address signer = _verify(info);
198     require(signer == signWallet, "not signed");
199
200     if (info.mintType == 0) {
201         require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
202         require(
203             wlLists[MintType.Whitelist][msg.sender] == true,
204             "not whitelist"
```

```
205            ); //wl
206            ethAmount = WHITELIST_PRICE;
207            mintType = MintType.Whitelist;
208        } else if (info.mintType == 1) {
209            require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
210            require(
211                wlLists[MintType.Whitelist][msg.sender] == true ||
212                    wlLists[MintType.Waitlist][msg.sender] == true,
213                "not waitlist"
214            );
215            ethAmount = WAITLIST_PRICE;
216            mintType = MintType.Waitlist;
217        } else if (info.mintType == 2) {
218            require(mintTotalCount[msg.sender].PbRound == 0, "Pb Minted");
219            ethAmount = MINT_PRICE;
220            mintType = MintType.Mint;
221        }
222
223        require(msg.value >= ethAmount, "Eth not enough.");
224        (bool sent, ) = adminWallet.call{value: msg.value}("");
225        require(sent, "Failed to send Ether");
226
227        loopGenToken(mintType, info.metadata);
228    }
```

Listing 5.1 The affected *genToken* function and its related dependencies

Recommendations

We recommend **adding the *nonce* (L53 in the code snippet 5.2) parameter to the *MintInfoI* struct. The nonce would prevent the minter from making the replay attack** since the payload will be for single use.

To use this *nonce*, we have to add the *genTokenNonces* mapping (L89) to track the spending of each signed payload, and the *nonce* must be updated every time the payload is used (L202).

Finally, when recovering the signer of the payload, the *nonce* has to be computed in the *_hash* function (L169 - 185).

CSKGen.sol

```
48  struct MintInfo {
49      address minter;
50      uint256 timestamp;
51      uint256 mintType;
52      uint256 metadata;
53      uint256 nonce;
54      bytes signature;
55  }
```

```
     // (...SNIPPED...)

 89  mapping(address => uint256) public genTokenNonces;

     // (...SNIPPED...)

169  function _hash(MintInfo memory info) internal view returns (bytes32) {
170      return
171          _hashTypedDataV4(
172              keccak256(
173                  abi.encode(
174                      keccak256(
175                          "MintInfo(address minter,uint256 timestamp,uint256
mintType,uint256 metadata, uint256 nonce)"
176                      ),
177                      info.minter,
178                      info.timestamp,
179                      info.mintType,
180                      info.metadata,
181                      info.nonce
182                  )
183              )
184          );
185  }
186
187  function _verify(MintInfo memory info) internal view returns (address) {
188      bytes32 digest = _hash(info);
189      return ECDSA.recover(digest, info.signature);
190  }
191
192
193  function genToken(MintInfo calldata info) external payable nonReentrant {
194      uint256 ethAmount;
195      MintType mintType;
196      require(block.timestamp <= info.timestamp + 1 minutes, "Times out");
197      require(info.minter == msg.sender, "not minter");
198
199      //verify
200      address signer = _verify(info);
201      require(signer == signWallet, "not signed");
202      require(info.nonce == genTokenNonces[info.minter]++, "Invalid nonce");
203
204      if (info.mintType == 0) {
205          require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
206          require(
207              wlLists[MintType.Whitelist][msg.sender] == true,
208              "not whitelist"
209          ); //wl
210          ethAmount = WHITELIST_PRICE;
211          mintType = MintType.Whitelist;
```

```
212     } else if (info.mintType == 1) {
213         require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
214         require(
215             wlLists[MintType.Whitelist][msg.sender] == true ||
216                 wlLists[MintType.Waitlist][msg.sender] == true,
217             "not waitlist"
218         );
219         ethAmount = WAITLIST_PRICE;
220         mintType = MintType.Waitlist;
221     } else if (info.mintType == 2) {
222         require(mintTotalCount[msg.sender].PbRound == 0, "Pb Minted");
223         ethAmount = MINT_PRICE;
224         mintType = MintType.Mint;
225     }
226 
227     require(msg.value >= ethAmount, "Eth not enough.");
228     (bool sent, ) = adminWallet.call{value: msg.value}("");
229     require(sent, "Failed to send Ether");
230 
231     loopGenToken(mintType, info.metadata);
232 }
```

Listing 5.2 The improved *genToken* function and its related dependencies

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 6	Possibly Bypassing The Condition To Generate Token		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol*		
Locations	*CSKGen.sol L: 190 - 228*		

Detailed Issue

The *genToken* function in the *CSKGen* contract allows users to mint NFTs, but **we discovered a vulnerability that could allow a user to bypass payment requirements, maximum minting limits per user, and whitelist checks.**

The root cause of all three cases is the same, **if the signer mistakenly sets the *MintInfo.mintType* value to a value outside the range of 0-2 during the signing process, causing the *genToken* function to skip all if-else conditions** (L200, L208, and L217 in code snippet 6.1). To provide more detail, we will explain each case separately:

1. **Bypass payment requirement**

   Since the *genToken* function bypasses all if-else conditions, **the *ethAmount* variable is not properly set to the minting price** (L191 in code snippet 6.1). This results in a default value of 0, which causes the function to bypass the payment verification (L223 in code snippet 6.1) even when the user has not paid any Ether. As a result, **the user is able to mint NFTs without paying any Ether.**

2. **Bypass maximum minting limits**

   Since the *genToken* function bypasses all if-else conditions, the minting limits set by the *CSKGen* contract are also bypassed. Specifically, the *CSKGen* contract allows each user to mint a maximum of two tokens, with one token per whitelist or waitlist round, and one token for the public round. However, with the vulnerability, **a user can mint more than 2 tokens because the *mintTotalCount[msg.sender].WlRound* and *mintTotalCount[msg.sender].PbRound* requirements are not checked** (L201, L209, L218 in code snippet 6.1), **allowing the user to bypass the intended minting limits.**

3. **Bypass whitelist checks**

   Since the *genToken* function bypasses all if-else conditions, the ***wlLists[MintType.Whitelist]*** and ***wlLists[MintType.Waitlist]*** states that are used to check whitelist or waitlist users are also skipped. This means that **users who are not on the whitelist or waitlist can still mint NFTs**, which could lead to an unfair distribution of tokens.

As a result, **any users can potentially bypass payment requirements, maximum minting limits, and whitelist checks when minting NFTs, leading to a loss of revenue for the system and unfair to other users.**

CSKGen.sol

```
48  struct MintInfo {
49      address minter;
50      uint256 timestamp;
51      uint256 mintType;
52      uint256 metadata;
53      bytes signature;
54  }

    // (...SNIPPED...)

190 function genToken(MintInfo calldata info) external payable nonReentrant {
191     uint256 ethAmount;
192     MintType mintType;
193     require(block.timestamp <= info.timestamp + 1 minutes, "Times out");
194     require(info.minter == msg.sender, "not minter");
195
196     //verify
197     address signer = _verify(info);
198     require(signer == signWallet, "not signed");
199
200     if (info.mintType == 0) {
201         require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
202         require(
203             wlLists[MintType.Whitelist][msg.sender] == true,
204             "not whitelist"
205         ); //wl
206         ethAmount = WHITELIST_PRICE;
207         mintType = MintType.Whitelist;
208     } else if (info.mintType == 1) {
209         require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
210         require(
211                 wlLists[MintType.Whitelist][msg.sender] == true ||
212                 wlLists[MintType.Waitlist][msg.sender] == true,
213                 "not waitlist"
214         );
215         ethAmount = WAITLIST_PRICE;
```

```
216         mintType = MintType.Waitlist;
217     } else if (info.mintType == 2) {
218         require(mintTotalCount[msg.sender].PbRound == 0, "Pb Minted");
219         ethAmount = MINT_PRICE;
220         mintType = MintType.Mint;
221     }
222 
223     require(msg.value >= ethAmount, "Eth not enough.");
224     (bool sent, ) = adminWallet.call{value: msg.value}("");
225     require(sent, "Failed to send Ether");
226 
227     loopGenToken(mintType, info.metadata);
228 }
```

Listing 6.1 The *genToken* function of the *CSKGen* contract

CSKGen.sol

```
152 function loopGenToken(MintType _mintType, uint256 metadata) internal {
153     string memory results = Strings.toString(metadata);
154 
155     if (_mintType == MintType.Mint) {
156         nftCore.mint(msg.sender, results, INFTCORE.MintType.Mint);
157         mintTotalCount[msg.sender].PbRound += 1;
158     } else if (_mintType == MintType.Whitelist) {
159         nftCore.mint(msg.sender, results, INFTCORE.MintType.Whitelist);
160         mintTotalCount[msg.sender].WlRound += 1;
161     } else if (_mintType == MintType.Waitlist) {
162         nftCore.mint(msg.sender, results, INFTCORE.MintType.Waitlist);
163         mintTotalCount[msg.sender].WlRound += 1;
164     }
165 }
```

Listing 6.2 The *loopGenToken* function of the *CSKGen* contract

Recommendations

We recommend improving the *genToken* function to handle the scenario where the *MintInfo.mintType* value is out of range by adding an else case that reverts the transaction if the *MintInfo.mintType* value is out of range (L221 - 223 in code snippet 6.3).

This improvement would prevent any unintended minting of NFTs and avoid the loss of revenue and unfairness that could occur if a user were able to bypass payment requirements, maximum minting limits per user, and whitelist checks.

CSKGen.sol

```
190 function genToken(MintInfo calldata info) external payable nonReentrant {
191     uint256 ethAmount;
192     MintType mintType;
193     require(block.timestamp <= info.timestamp + 1 minutes, "Times out");
194     require(info.minter == msg.sender, "not minter");
195
196     //verify
197     address signer = _verify(info);
198     require(signer == signWallet, "not signed");
199
200     if (info.mintType == 0) {
201         require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
202         require(
203             wlLists[MintType.Whitelist][msg.sender] == true,
204             "not whitelist"
205         ); //wl
206         ethAmount = WHITELIST_PRICE;
207         mintType = MintType.Whitelist;
208     } else if (info.mintType == 1) {
209         require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
210         require(
211                 wlLists[MintType.Whitelist][msg.sender] == true ||
212                 wlLists[MintType.Waitlist][msg.sender] == true,
213                 "not waitlist"
214         );
215         ethAmount = WAITLIST_PRICE;
216         mintType = MintType.Waitlist;
217     } else if (info.mintType == 2) {
218         require(mintTotalCount[msg.sender].PbRound == 0, "Pb Minted");
219         ethAmount = MINT_PRICE;
220         mintType = MintType.Mint;
221     } else {
222         revert("Incorrect Minting Type: Out Of Range");
223     }
224
225     require(msg.value >= ethAmount, "Eth not enough.");
226     (bool sent, ) = adminWallet.call{value: msg.value}("");
227     require(sent, "Failed to send Ether");
228
229     loopGenToken(mintType, info.metadata);
230 }
```

Listing 6.3 The improved *genToken* function of the *CSKGen* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 7	Non-Uniqueness NFT Metadata Assignment		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/CSKNFT.sol*		
Locations	*CSKNFT.sol* • *L: 86* • *L: 214 - 223 (The delMint function)* • *L: 225 - 261 (The mint function)* • *L: 277 - 303 (The updateFlagStatus function)*		

Detailed Issue

The *metadata* within the *UserAsset* struct (L86 in code snippet 7.1) is the crucial part of the *CSKNFT* contract that represents the NFT's properties and rarity.

For explanation, the metadata of NFT is determined once minted (code snippet 7.2) and could be changed via the *updateFlagStatus* function (code snippet 7.3).

However, these operations fully trust that **the metadata provided by the off-chain service (the *CodeSekai* platform) is legitimate without verifying that each metadata must be assigned to only a specific NFT.** In other words, each single metadata should ideally be assigned to only a single NFT at a time.

If the same metadata is assigned to multiple NFTs, several issues and concerns–such as transparency, traceability, fairness, uniqueness, etc–**can occur due to the non-uniqueness of the in-game NFT items' metadata.**

CSKNFT.sol

```
83  struct UserAsset {
84      uint256 tokenId;
85      bool isAvailable;
86      string metadata;
87  }
```

Listing 7.1 The *metadata* that represents the NFT's properties and rarity

CSKNFT.sol

```
214 function delMint(address _userAddr, string memory metadata) internal {
215     //start tokenId at 1
216     tokenIdCounter.increment();
217
218     uint256 tokenId = tokenIdCounter.current();
219     require(tokenId <= TOTAL_SUPPLY, "Max supply");
220
221     _safeMint(_userAddr, tokenId);
222     tokenInfo[tokenId] = UserAsset(tokenId, true, metadata);
223 }
224
225 function mint(
226     address _userAddr,
227     string calldata metadata,
228     MintType _mintType
229 ) external onlyRole(MINTER_ROLE) {
230     if (_mintType == MintType.Mint) {
231         require(
232             block.timestamp >= mintDates.MINT_START_DATE,
233             "not started."
234         );
235         require(block.timestamp <= mintDates.MINT_END_DATE, "ended.");
236     } else if (_mintType == MintType.Whitelist) {
237         require(
238             block.timestamp >= mintDates.START_WHITELIST,
239             "Wl not started."
240         );
241         require(block.timestamp <= mintDates.END_WHITELIST, "Wl ended.");
242     } else if (_mintType == MintType.Waitlist) {
243         require(
244             block.timestamp >= mintDates.START_WAITLIST,
245             "Waitlist not started."
246         );
247         require(
248             block.timestamp <= mintDates.END_WAITLIST,
249             "Waitlist ended."
250         );
251     }
252
253     delMint(_userAddr, metadata);
254
255     emit MintNft(
256         _userAddr,
257         tokenIdCounter.current(),
258         block.timestamp,
259         _mintType
260     );
261 }
```

Listing 7.2 The *mint* function that trusts the provided *metadata*

CSKNFT.sol

```
277 function updateFlagStatus(SignInfo calldata _info)
278     external
279     payable
280     nonReentrant
281 {
282     require(ownerOf(_info.tokenId) == msg.sender, "Not Owner.");
283     require(
284         tokenInfo[_info.tokenId].isAvailable != _info.status,
285         "Same status."
286     );
287
288     //verify
289     address signer = _verify(_info);
290     require(signer == signWallet, "not signed");
291
292     if (_info.status) {
293         require(msg.value >= PORTAL_PRICE, "Eth not enough.");
294
295         (bool sent, ) = adminWallet.call{value: msg.value}("");
296         require(sent, "Failed send");
297         tokenInfo[_info.tokenId].metadata = _info.metadata;
298     }
299
300     tokenInfo[_info.tokenId].isAvailable = _info.status;
301
302     emit ChangeItemStatus(msg.sender, _info);
303 }
```

Listing 7.3 The *updateFlagStatus* function that trusts the provided *metadata*

Recommendations

Since the *CSKNFT* contract is designed to fully trust the metadata provided by the off-chain service (the *CodeSekai* platform), **no recommended code can fully fix this issue without breaking the contract's features.**

However, we recommend the *CodeSekai* team to **redesign and reimplement both the related on-chain service (smart contract) and off-chain service (front-end and back-end) to guarantee that the case of the non-uniqueness of the in-game NFT items' metadata described above would not be happened unexpectedly as well as not compromising the platform's business requirements.**

One recommended mitigation strategy is to perform a source code review on the off-chain services to ensure that the generated metadata for NFT items are unique.

Reassessment

The *CodeSekai* team has acknowledged the existence of this issue and already implemented a solution in the form of a smart contract called ***RandomWorker***. This contract is responsible for generating and storing trackable metadata transparently. To prevent unauthorized access by hackers, the contract must remain enclosed until the minting process is complete.

Furthermore, it is important to note that the ***RandomWorker*** contract falls outside the scope of our audit. Therefore, we strongly recommend that the team undertake a full security audit of the ***RandomWorker*** contract.

No. 8	Trust And Fairness Of Metadata Generation		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/CSKGen.sol* *contracts/CSKNFT.sol*		
Locations	*CSKGen.sol L: 152 - 165 (*The *loopGenToken* function*)* *CSKNFT.sol L: 152 - 165 (*The *updateFlagStatus* function*)*		

Detailed Issue

The *genToken* and *updateFlagStatus* functions utilize signed payloads (*MintInfo* for the *genToken* function and *SignInfo* for the *updateFlagStatus* function) to mint (L153 and L227 in the code snippet 8.1) and update tokens (L297 in the code snippet 8.1), respectively. Both payload types include a crucial field, namely the *metadata* field.

However, **it is important to note that in the minting and updating process, the *metadata* is assigned directly from the off-chain source**. For this reason, the current audit scope cannot guarantee the randomness of the *metadata*, as its accuracy and reliability depend on the off-chain source.

This creates a potential security risk and could compromise the integrity and reliability of the token's *metadata*.

CSKGen.sol

```
152 function loopGenToken(MintType _mintType, uint256 metadata) internal {
153     string memory results = Strings.toString(metadata);
154
155     if (_mintType == MintType.Mint) {
156         nftCore.mint(msg.sender, results, INFTCORE.MintType.Mint);
157         mintTotalCount[msg.sender].PbRound += 1;
158     } else if (_mintType == MintType.Whitelist) {
159         nftCore.mint(msg.sender, results, INFTCORE.MintType.Whitelist);
160         mintTotalCount[msg.sender].WlRound += 1;
161     } else if (_mintType == MintType.Waitlist) {
162         nftCore.mint(msg.sender, results, INFTCORE.MintType.Waitlist);
163         mintTotalCount[msg.sender].WlRound += 1;
164     }
165 }

    // (...SNIPPED...)
```

```
190 function genToken(MintInfo calldata info) external payable nonReentrant {
191     uint256 ethAmount;
192     MintType mintType;
193     require(block.timestamp <= info.timestamp + 1 minutes, "Times out");
194     require(info.minter == msg.sender, "not minter");
195
196     //verify
197     address signer = _verify(info);
198     require(signer == signWallet, "not signed");
199
200     if (info.mintType == 0) {
201         require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
202         require(
203             wlLists[MintType.Whitelist][msg.sender] == true,
204             "not whitelist"
205         ); //wl
206         ethAmount = WHITELIST_PRICE;
207         mintType = MintType.Whitelist;
208     } else if (info.mintType == 1) {
209         require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
210         require(
211             wlLists[MintType.Whitelist][msg.sender] == true ||
212                 wlLists[MintType.Waitlist][msg.sender] == true,
213             "not waitlist"
214         );
215         ethAmount = WAITLIST_PRICE;
216         mintType = MintType.Waitlist;
217     } else if (info.mintType == 2) {
218         require(mintTotalCount[msg.sender].PbRound == 0, "Pb Minted");
219         ethAmount = MINT_PRICE;
220         mintType = MintType.Mint;
221     }
222
223     require(msg.value >= ethAmount, "Eth not enough.");
224     (bool sent, ) = adminWallet.call{value: msg.value}("");
225     require(sent, "Failed to send Ether");
226
227     loopGenToken(mintType, info.metadata);
228 }
```

Listing 8.1 The *loopGenToken* and *genToken* functions of the *CSKNFT* contract

CSKNFT.sol

```
277 function updateFlagStatus(SignInfo calldata _info)
278     external
279     payable
280     nonReentrant
281 {
282     require(ownerOf(_info.tokenId) == msg.sender, "Not Owner.");
283     require(
284         tokenInfo[_info.tokenId].isAvailable != _info.status,
285         "Same status."
286     );
287
288     //verify
289     address signer = _verify(_info);
290     require(signer == signWallet, "not signed");
291
292     if (_info.status) {
293         require(msg.value >= PORTAL_PRICE, "Eth not enough.");
294
295         (bool sent, ) = adminWallet.call{value: msg.value}("");
296         require(sent, "Failed send");
297         tokenInfo[_info.tokenId].metadata = _info.metadata;
298     }
299
300     tokenInfo[_info.tokenId].isAvailable = _info.status;
301
302     emit ChangeItemStatus(msg.sender, _info);
303 }
```

Listing 8.2 The *updateFlagStatus* function of the *CSKNFT* contract

Recommendations

We strongly recommend that the team conducts a thorough **penetration testing and source code review of the off-chain source**. This testing will ensure the authenticity and reliability of the off-chain source before assigning metadata to tokens in the smart contract, thereby increasing the reliability and integrity of the token's metadata.

Reassessment

The *CodeSekai* team has acknowledged the existence of this issue and already implemented a solution in the form of a smart contract called ***RandomWorker***. This contract is responsible for generating and storing trackable metadata transparently. To prevent unauthorized access by hackers, the contract must remain enclosed until the minting process is complete.

Furthermore, it is important to note that the ***RandomWorker*** contract falls outside the scope of our audit. Therefore, we strongly recommend that the team undertake a full security audit of the ***RandomWorker*** contract.

No. 9	Recommended Improving Transparency And Trustworthiness		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*@openzeppelin/contracts/access/AccessControl.sol* *contracts/CSKGen.sol* *contracts/CSKNFT.sol*		
Locations	*AccessControl.sol* ● *L: 144 - 146 (*The *grantRole* function*)* ● *L: 159 - 161 (*The *revokeRole* function*)* *CSKGen.sol* ● *L: 96 - 101 (*The *setRandomWorker* function*)* ● *L: 103 - 108 (*The *setSingWallet* function*)* ● *L: 110 - 121 (*The *setPrice* function*)* ● *L: 132 - 150 (*The *setWhitelists* function*)* *CSKNFT.sol* ● *L: 133 - 138 (*The *setBaseURI* function*)* ● *L: 197 - 212 (*The *setPeriods* function*)* ● *L: 270 - 275 (*The *setPortalPrice* function*)*		

Detailed Issue

We have discovered that the *CSKGen* and *CSKNFT* contracts utilize the *AccessControl* contract, inheriting it to implement high-level roles necessary for performing critical mechanisms.

The following lists all privileged functions that should be improved transparency and trustworthiness:

- The ***AccessControl*** contract
    - The *grantRole* function
    - The *revokeRole* function

- The ***CSKGen*** contract
    - The *setRandomWorker* function
    - The *setSingWallet* function
    - The *setPrice* function
    - The *setWhitelists* function

- The ***CSKNFT*** contract
  - The *setBaseURI* function
  - The *setPeriods* function
  - The *setPortalPrice* function

Our analysis found that those functions listed can change important states, which could affect the users' assets. For this reason, we consider that those functions should be improved for transparency and trustworthiness.

CSKGen.sol

```
103 function setSingWallet(address _newSignWallet)
104     public
105     onlyRole(DEV_ROLE)
106 {
107     signWallet = _newSignWallet;
108 }
```

Listing 9.1 The example function that can be invoked by a privileged role

Code snippet 9.1 above exhibits an example of the privileged function *setSingWallet*. The function can be executed by an admin with the *DEV_ROLE* role (L105).

Recommendations

We recommend governing the associated setter functions with the ***Timelock*** mechanism to improve the transparency and trustworthiness of the privileged operations.

We recommend **replacing the *DEV_ROLE* role with the *TIMELOCK_DEV_ROLE* role instead. The *TIMELOCK_DEV_ROLE* role should point to an address of the *Timelock* contract**.

The ***Timelock*** mechanism allows for a specified amount of time to pass before the proposed changes can be executed, providing users with sufficient time to review and assess the proposed changes.

For the recommended ***Timelock*** contract, please refer to: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/governance/TimelockController.sol.

Please apply the *TIMELOCK_DEV_ROLE* to the following functions:

- The ***AccessControl*** contract
  - The *grantRole* function
  - The *revokeRole* function

- The **CSKGen** contract
  - The *setRandomWorker* function
  - The *setSingWallet* function
  - The *setPrice* function
  - The *setWhitelists* function

- The **CSKNFT** contract
  - The *setBaseURI* function
  - The *setPeriods* function
  - The *setPortalPrice* function

CSKGen.sol

```
     bytes32 public constant TIMELOCK_DEV_ROLE = keccak256("TIMELOCK_DEV_ROLE");
     address public timelockAddress;

     // (...SNIPPED...)

     event SetTimelock(address indexed prevTimelockAddress, address indexed
     newTimeLockAddress);

     // (...SNIPPED...)

74   constructor(
75       address _nftCore,
76       address _randomWokerAddr,
77       address payable _adminWallet,
78       address _signWallet,
79       address _timelockAddress
80   ) EIP712(SIGNING_DOMAIN, SIGNATURE_VERSION) {
81       require(_timelockAddress != address(0), "Invalid address");
82       _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
83       _grantRole(TIMELOCK_DEV_ROLE, _timelockAddress); // assigning a role to the
     timelock contract
84       nftCore = INFTCORE(_nftCore);
85       iRandomWorker = IRANDOMWORKER(_randomWokerAddr);
86       adminWallet = _adminWallet;
87       signWallet = _signWallet;
88       timelockAddress = _timelockAddress;
87   }

     // (...SNIPPED...)

105  function setSingWallet(address _newSignWallet)
106      public
107      onlyRole(TIMELOCK_DEV_ROLE)
108  {
109      signWallet = _newSignWallet;
```

```
110 }

    // (...SNIPPED...)

232 function grantRole(bytes32 role, address account)
233     public
234     virtual
235     override(AccessControl)
236     onlyRole(TIMELOCK_DEV_ROLE)
237 {
238     _grantRole(role, account);
239 }
240
241 function setTimelock(address newTimelockAddress) external
    onlyRole(TIMELOCK_DEV_ROLE) {
242     require(newTimelockAddress != address(0), "Invalid newTimelockAddress
    address");
243     address prevTimelockAddress = timelockAddress;
244     timelockAddress = newTimelockAddress;
245
246     _revokeRole(TIMELOCK_DEV_ROLE, prevTimelockAddress);
247     _grantRole(TIMELOCK_DEV_ROLE, newTimelockAddress);
248
249     emit SetTimelock(prevTimelockAddress, newTimelockAddress);
250 }
```

Listing 9.2 The example using of *Timelock* mechanism in the *CSKGen* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 10	Burning Tokens Without Validating Availability Flag		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKNFT.sol* *@openzeppelin/contracts/token/ERC721/extensions/ERC721Burnable.sol*		
Locations	*ERC721Burnable.sol L: 21 -25*		

Detailed Issue

We found that the *CSKNFT* contract inherits the *ERC721Burnable* contract, which includes a *burn* function for burning tokens. However, this function does not check the ***tokenInfo[tokenId].isAvailable*** flag status state, meaning that unavailable tokens can still be burned, even though they should not be burnable or transferable.

As a result, **if a token owner mistakenly invokes the *burn* function for an unavailable token, the token will be burned and may lead to an inconsistency between the off-chain and on-chain data that is associated with the token**.

CSKNFT.sol

```
1  // SPDX-License-Identifier: CODESEKAI
2  pragma solidity ^0.8.0;
3
4  import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
5  import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
6  import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Burnable.sol";
   // (...SNIPPED...)

15 contract CSKNFT is
16     ERC721,
17     EIP712,
18     ERC721Enumerable,
19     ERC721Burnable,
20     AccessControl,
21     Ownable,
22     Initializable,
23     ReentrancyGuard
24 {
       // (...SNIPPED...)
```

```
322 }
```

Listing 10.1 The *CSKNFT* contract that inherits the *ERC721Burnable* contract

ERC721Burnable.sol

```
21  function burn(uint256 tokenId) public virtual {
22      //solhint-disable-next-line max-line-length
23      require(_isApprovedOrOwner(_msgSender(), tokenId), "ERC721: caller is not
    token owner or approved");
24      _burn(tokenId);
25  }
```

Listing 10.2 The *burn* function of the *ERC721Burnable* contract

Recommendations

We recommend overriding the *burn* function in the *CSKNFT* contract to include a check of the ***tokenInfo[tokenId].isAvailable*** flag status state before the token is burned. This will **ensure that unavailable tokens are not burned**, preventing the inconsistency that can occur between the off-chain and on-chain data.

CSKNFT.sol

```
323 function burn(uint256 tokenId) public override {
324     require(checkFlagStatus(tokenId), "not available");
325     super.burn(tokenId);
326 }
```

Listing 10.3 The overridden *burn* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 11	Improper Verification Of Supply Checking		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKNFT.sol* *@openzeppelin/contracts/token/ERC721/extensions/ERC721Burnable.sol* *@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol*		
Locations	*CSKNFT.sol L: 214 - 223*		

Detailed Issue

The *CSKNFT* contract uses the *tokenIdCounter* variable to keep track of the number of tokens minted. When a new token is minted using the *delMint* function, the *tokenIdCounter* is incremented, and the *tokenId* of the new token is set to the current value of the *tokenIdCounter*. The *tokenId* is then checked against the *TOTAL_SUPPLY* constant to ensure that the total supply of tokens has not been exceeded (L219 in code snippet 11.1).

However, the contract relies on the *tokenId* to check with the *TOTAL_SUPPLY* constant, rather than querying the actual total supply of tokens from the contract using the *ERC721Enumerable.totalSupply* function. **This could cause the *tokenId* to exceed the actual total supply of tokens, resulting in the *delMint* function failing to mint a new token and reverting the transaction.**

For example, the *TOTAL_SUPPLY* constant is set to 5555, and suppose 5555 tokens have already been minted. If a user burns their token, it will decrease the current number of tokens in circulation to 5554.

Subsequently, if another user tries to mint a new token using the *delMint* function, the *tokenIdCounter* will be incremented to 5556. This will cause the function to fail because the *tokenId* of the new token will exceed the *TOTAL_SUPPLY* constant, leading to confusion and potentially causing failed transactions.

As a result, this issue could cause confusion and potentially lead to failed transactions if users try to mint new tokens when the actual total supply of tokens is less than the *TOTAL_SUPPLY* constant.

CSKNFT.sol

```
37   Counters.Counter private tokenIdCounter;

     // (...SNIPPED...)

58   uint256 public constant TOTAL_SUPPLY = 5_555;

     // (...SNIPPED...)

214  function delMint(address _userAddr, string memory metadata) internal {
215      //start tokenId at 1
216      tokenIdCounter.increment();
217
218      uint256 tokenId = tokenIdCounter.current();
219      require(tokenId <= TOTAL_SUPPLY, "Max supply");
220
221      _safeMint(_userAddr, tokenId);
222      tokenInfo[tokenId] = UserAsset(tokenId, true, metadata);
223  }
```

Listing 11.1 The *delMint* function of the *CSKNFT* contract

Recommendations

We recommend **using the *totalSupply* function provided by the *ERC721Enumerable* contract that gives the actual total number of tokens in circulation to check if the total supply has been reached instead of relying on the *tokenId* value**.

This is because relying on *tokenId* can lead to issues when tokens are burned, and the *tokenId* value exceeds the *TOTAL_SUPPLY* constant.

CSKNFT.sol

```
214  function delMint(address _userAddr, string memory metadata) internal {
215      uint256 currentSupply = totalSupply();
216      require(currentSupply < TOTAL_SUPPLY, "Max supply");
217
218      //start tokenId at 1
219      tokenIdCounter.increment();
220      uint256 tokenId = tokenIdCounter.current();
221
222      _safeMint(_userAddr, tokenId);
223      tokenInfo[tokenId] = UserAsset(tokenId, true, metadata);
224  }
```

Listing 11.2 The improved *delMint* function of the *CSKNFT* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 12	Inconsistent State In Token Management When Burning Tokens		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKNFT.sol* *@openzeppelin/contracts/token/ERC721/extensions/ERC721Burnable.sol*		
Locations	*CSKNFT.sol L: 83 - 87, 90*		

Detailed Issue

The *CSKNFT* contract uses the *tokenInfo* mapping to keep track of the token information for each token. The mapping maps a token ID to a *UserAsset* struct, which contains the token ID, a boolean flag indicating whether the token is available for transfer, and the metadata associated with the token (L90 in code snippet 12.1).

However, when a user burns a token using the *ERC721Burnable.burn* function, **the contract does not update the *tokenInfo* mapping to reflect the fact that the token has been burned**. This means that **the *tokenId*, *isAvailable* flag, and *metadata* associated with the burned token continue to hold their old values, even though the token no longer exists.**

This inconsistency can cause issues in several functions that rely on the *tokenInfo* mapping.

1. The *checkFlagStatus* function (L144 - 146 in code snippet 12.2)
2. The *checkMetadata* function (L148 - 154 in code snippet 12.2)
3. The *getUserTokenAndInfos* function (L177 - 195 in code snippet 12.2)

As a result, these functions can return inconsistent or incorrect results when querying for information about a burned token.

CSKNFT.sol

```
83 struct UserAsset {
84     uint256 tokenId;
85     bool isAvailable;
86     string metadata;
87 }
88
89 MintDates private mintDates;
```

```
90  mapping(uint256 => UserAsset) private tokenInfo;
```

Listing 12.1 The *tokenInfo* state variable

CSKNFT.sol

```
144 function checkFlagStatus(uint256 tokenId) public view returns (bool) {
145     return tokenInfo[tokenId].isAvailable;
146 }
147
148 function checkMetadata(uint256 tokenId)
149     public
150     view
151     returns (string memory)
152 {
153     return tokenInfo[tokenId].metadata;
154 }

    // (...SNIPPED...)

177 function getUserTokenAndInfos(address userAddress)
178     public
179     view
180     returns (UserAsset[] memory)
181 {
182     uint256 balance = balanceOf(userAddress);
183     UserAsset[] memory userAssets = new UserAsset[](balance);
184
185     for (uint32 i = 0; i < uint32(balance); i++) {
186         uint256 tokenId = tokenOfOwnerByIndex(userAddress, i);
187         string memory metadata = tokenInfo[tokenId].metadata;
188         bool status = checkFlagStatus(tokenId);
189
190         userAssets[i].tokenId = tokenId;
191         userAssets[i].isAvailable = status;
192         userAssets[i].metadata = metadata;
193     }
194     return userAssets;
195 }
```

Listing 12.2 The several functions that rely on the *tokenInfo* mapping

Recommendations

We recommend **overriding the *ERC721Burnable.burn* function and adding a condition to check if the token is available before deleting the token information from the *tokenInfo* mapping**. This will ensure that the token information is properly updated when a token is burned

CSKNFT.sol

```
    event BurnNFT(uint256 indexed tokenId, address indexed burner, uint256
    burnedAt);

323 function burn(uint256 tokenId) public override {
324     require(checkFlagStatus(tokenId), "not available");
325     emit BurnNFT(tokenId, msg.sender, block.timestamp);
326     delete tokenInfo[tokenId];
327     super.burn(tokenId);
328 }
```

Listing 12.3 The overridden *ERC721Burnable.burn* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 13	Incorrect Condition For Removing Whitelist		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol*		
Locations	*CSKGen.sol L: 132 - 150 (*The *setWhitelists* function*)*		

Detailed Issue

The *CSKGen* contract includes a feature to manage the whitelisted addresses through the *setWhitelists* function. However, We discovered that there is an incorrect functionality in the remove whitelist logic (L143 - 149 in the provided code snippet).

To elaborate, the **incorrect logic is verifying the statement *!wlLists[wlType][_userAddresses[i]]*, which evaluates the false boolean value rather than true in order to remove the given addresses from the whitelist**.

This issue could lead to unexpected results and must be addressed to ensure proper functionality.

CSKGen.sol

```
132 function setWhitelists(
133     MintType wlType,
134     address[] memory _userAddresses,
135     Whitelist _doType
136 ) public onlyRole(DEV_ROLE) {
137     if (_doType == Whitelist.Add) {
138         for (uint32 i = 0; i < _userAddresses.length; i++) {
139             if (!wlLists[wlType][_userAddresses[i]]) {
140                 wlLists[wlType][_userAddresses[i]] = true;
141             }
142         }
143     } else if (_doType == Whitelist.Remove) {
144         for (uint32 i = 0; i < _userAddresses.length; i++) {
145             if (!wlLists[wlType][_userAddresses[i]]) {
146                 wlLists[wlType][_userAddresses[i]] = false;
147             }
148         }
149     }
150 }
```

Listing 13.1 The *setWhitelists* function of the *CSKGen* contract

Recommendations

We recommend changing the logic for disabling whitelist addresses as shown in the code snippet below.

CSKGen.sol

```
132 function setWhitelists(
133     MintType wlType,
134     address[] memory _userAddresses,
135     Whitelist _doType
136 ) public onlyRole(DEV_ROLE) {
137     if (_doType == Whitelist.Add) {
138         for (uint32 i = 0; i < _userAddresses.length; i++) {
139             if (!wlLists[wlType][_userAddresses[i]]) {
140                 wlLists[wlType][_userAddresses[i]] = true;
141             }
142         }
143     } else if (_doType == Whitelist.Remove) {
144         for (uint32 i = 0; i < _userAddresses.length; i++) {
145             if (wlLists[wlType][_userAddresses[i]]) {
146                 wlLists[wlType][_userAddresses[i]] = false;
147             }
148         }
149     }
150 }
```

Listing 13.2 The improved *setWhitelists* function of the *CSKGen* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team fixed this issue as per our suggestion.

No. 14	No Upper Bound For The Portal Price		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKNFT.sol*		
Locations	*CSKNFT.sol L: 270 - 276*		

Detailed Issue

The *updateFlagStatus* function of the *CSKNFT* contract allows the NFT owner to update an NFT's metadata and status by paying a fee greater or equal to the *PORTAL_PRICE* (L293 in code snippet 14.1).

However, we found that the *setPortalPrice* function allows the *DEV_ROLE* to arbitrarily adjust the *PORTAL_PRICE* variable (L270 - 275 in code snippet 14.2) without the maximum upper bound of the portal fee, which directly affects the users' fee when updating their NFT.

Consider the case that the fee is set too high, the users might not be able to bridge back to the on-chain service.

CSKNFT.sol

```
59   uint256 public PORTAL_PRICE = 0.0005 ether;

     //(...SNIPPED...)

277  function updateFlagStatus(SignInfo calldata _info)
278      external
279      payable
280      nonReentrant
281  {
282      require(ownerOf(_info.tokenId) == msg.sender, "Not Owner.");
283      require(
284          tokenInfo[_info.tokenId].isAvailable != _info.status,
285          "Same status."
286      );
287
288      //verify
289      address signer = _verify(_info);
290      require(signer == signWallet, "not signed");
```

```
291
292     if (_info.status) {
293         require(msg.value >= PORTAL_PRICE, "Eth not enough.");
294
295         (bool sent, ) = adminWallet.call{value: msg.value}("");
296         require(sent, "Failed send");
297         tokenInfo[_info.tokenId].metadata = _info.metadata;
298     }
299
300     tokenInfo[_info.tokenId].isAvailable = _info.status;
301
302     emit ChangeItemStatus(msg.sender, _info);
303 }
```

Listing 14.1 The *updateFlagStatus* function that
charges a fee depending on the *PORTAL_PRICE* variable

CSKNFT.sol

```
270 function setPortalPrice(uint256 newPrice)
271     public
272     onlyRole(DEV_ROLE)
273 {
274     PORTAL_PRICE = newPrice;
275 }
```

Listing 14.2 The *setPortalPrice* function lacks of boundary checking for the portal price

Recommendations

We recommend setting the maximum upper bound of the portal fee in the *setPortalPrice* function as shown in the code snippet below.

The ***MAX_PORTAL_PRICE*** should be a constant to limit the maximum users' fee when updating their NFT.

CSKNFT.sol

```
    // set max portal price properly
    uint256 public constant MAX_PORTAL_PRICE = 0.0005 ether;

    //(...SNIPPED...)

270 function setPortalPrice(uint256 newPrice)
271     public
272     onlyRole(DEV_ROLE)
273 {
274     require(newPrice_ <= MAX_PORTAL_PRICE, "invalid portal price");
```

```
275        PORTAL_PRICE = newPrice;
276    }
```

Listing 14.3 The improved *setPortalPrice* function that checks the maximum portal price

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 15	Lack Of Setter Function For adminWallet State		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol*		
Locations	*CSKGen.sol L: 36*		

Detailed Issue

We found that the *CSKGen* contract has a state variable called *adminWallet* (L36 in code snippet 15.1), which is used to receive funds when users mint NFT tokens (L224 in code snippet 15.1).

We noticed that the *adminWallet* wallet is only set once in the constructor and cannot be changed later because there is no setter function in the contract. This means that **if the *adminWallet* wallet is compromised by an attacker or lost due to unforeseen circumstances, the owner of the *adminWallet* wallet would not be able to receive any revenue generated by the NFT minting process**.

CSKGen.sol

```
33   contract CSKGen is EIP712, AccessControl, ReentrancyGuard {
34       INFTCORE public nftCore;
35       IRANDOMWORKER private iRandomWorker;
36       address payable private adminWallet;

         // (...SNIPPED...)

190      function genToken(MintInfo calldata info) external payable nonReentrant {

             // (...SNIPPED...)

223          require(msg.value >= ethAmount, "Eth not enough.");
224          (bool sent, ) = adminWallet.call{value: msg.value}("");
225          require(sent, "Failed to send Ether");
226
227          loopGenToken(mintType, info.metadata);
228      }
229  }
```

Listing 15.1 The *adminWallet* wallet of the *CSKGen* contract

Recommendations

We recommend **adding a setter function for the *adminWallet* wallet in the *CSKGen* contract**.

This function should only be accessible to the *DEFAULT_ADMIN_ROLE* to prevent unauthorized changes and ensure that the *adminWallet* wallet can be updated in case of compromise or loss, and that any revenue generated by the NFT minting process can be received.

CSKGen.sol

```
230 function setAdminWallet(address payable _adminWallet)
231     public
232     onlyRole(DEFAULT_ADMIN_ROLE)
233 {
234     require(_adminWallet != address(0), "Invalid address");
235     address previousAdminWallet = adminWallet;
236     adminWallet = _adminWallet;
237     emit setAdminWallet(previousAdminWallet, adminWallet);
238 }
```

Listing 15.2 The *setAdminWallet* function of the *CSKGen* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 16	Directly Minting Without Permission		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKNFT.sol*		
Locations	*CSKNFT.sol L: 225 - 261 (*The *mint* function*)*		

Detailed Issue

The *CSKNFT* contract uses the *AccessControl* contract to implement high-level roles required for performing critical mechanisms, with each privilege role allowing more than one address to hold (code snippet 16.2).

We observed that **the *mint* function allows accounts holding the *MINTER_ROLE* to execute it** (L229 in the code snippet 16.1). However, we also identified **a potential risk that the *DEAFULT_ADMIN_ROLE* can mistakenly assign the *MINTER_ROLE* to an incorrect account that was not intended to have it**.

Consequently, if that account directly invokes the *mint* function, it would result in the direct minting of the token without the required permission and lead to unexpected consequences.

CSKNFT.sol

```
225  function mint(
226      address _userAddr,
227      string calldata metadata,
228      MintType _mintType
229  ) external onlyRole(MINTER_ROLE) {
230      if (_mintType == MintType.Mint) {
231          require(
232              block.timestamp >= mintDates.MINT_START_DATE,
233              "not started."
234          );
235          require(block.timestamp <= mintDates.MINT_END_DATE, "ended.");
236      } else if (_mintType == MintType.Whitelist) {
237          require(
238              block.timestamp >= mintDates.START_WHITELIST,
239              "Wl not started."
240          );
241          require(block.timestamp <= mintDates.END_WHITELIST, "Wl ended.");
242      } else if (_mintType == MintType.Waitlist) {
243          require(
```

```
244                 block.timestamp >= mintDates.START_WAITLIST,
245                 "Waitlist not started."
246             );
247             require(
248                 block.timestamp <= mintDates.END_WAITLIST,
249                 "Waitlist ended."
250             );
251         }
252 
253         delMint(_userAddr, metadata);
254 
255         emit MintNft(
256             _userAddr,
257             tokenIdCounter.current(),
258             block.timestamp,
259             _mintType
260         );
261 }
```

Listing 16.1 The *mint* function of the *CSKNFT* contract

AccessControl.sol

```
50 struct RoleData {
51     mapping(address => bool) members;
52     bytes32 adminRole;
53 }
54 
55 mapping(bytes32 => RoleData) private _roles;
```

Listing 16.2 The *RoleData* struct of the *AccessControl* contract

Recommendations

We recommend **introducing the *cskGen* address variable to store the *CSKGen* contract address along with its setter function (the *setCSKGen* function) and restricting the *mint* function to only be called by the *CSKGen* contract** as shown in the code snippet below.

Furthermore, we suggest governing the *setCSKGen* function with the *TIMELOCK_DEV_ROLE*. The *TIMELOCK_DEV_ROLE* is assigned as the only role authorized to execute the associated functions. This would improve the transparency and trustworthiness of privileged operations.

For more information about the usage of the *TIMELOCK_DEV_ROLE*, please refer to issue #9 - *Recommended Improvements for Transparency and Trustworthiness.*

CSKNFT.sol

```
31  address public cskGen;

    // (...SNIPPED...)

40  constructor()
41      ERC721("CodeSekaiNFT", "CSKI")
42      EIP712(SIGNING_DOMAIN, SIGNATURE_VERSION)
43  {
44      _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
45      _grantRole(DEV_ROLE, msg.sender);
46  }

    // (...SNIPPED...)

227 function mint(
228     address _userAddr,
229     string calldata metadata,
230     MintType _mintType
231 ) external {
232     require(cskGen == _msgSender(), "permission denied");
233
234     if (_mintType == MintType.Mint) {
235         require(
236             block.timestamp >= mintDates.MINT_START_DATE,
237             "not started."
238         );
239         require(block.timestamp <= mintDates.MINT_END_DATE, "ended.");
240     } else if (_mintType == MintType.Whitelist) {
241         require(
242             block.timestamp >= mintDates.START_WHITELIST,
243             "Wl not started."
244         );
243         require(block.timestamp <= mintDates.END_WHITELIST, "Wl ended.");
246     } else if (_mintType == MintType.Waitlist) {
247         require(
248             block.timestamp >= mintDates.START_WAITLIST,
249             "Waitlist not started."
250         );
251         require(
252             block.timestamp <= mintDates.END_WAITLIST,
253             "Waitlist ended."
254         );
255     }
256
257     delMint(_userAddr, metadata);
258
259     emit MintNft(
260         _userAddr,
261         tokenIdCounter.current(),
```

```
262            block.timestamp,
263            _mintType
264        );
265    }

       // (...SNIPPED...)

281    function setCSKGen(uint256 newCSKGen)
282        public
283        onlyRole(TIMELOCK_DEV_ROLE)
284    {
285        require(newCSKGen != address(0), "Invalid address");
286        address prevCSKGen = cskGen;
287        cskGen = newCSKGen;
288        emit SetCSKGen(prevCSKGen, newCSKGen);
289    }
```

Listing 16.3 The improved *mint* function and the associated functions.

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 17	Possibly Setting Improper Period For Each Minting Round		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKNFT.sol*		
Locations	*CSKNFT.sol L: 197 - 212*		

Detailed Issue

The *CSKNFT* contract has separate periods ***Whitelist***, ***Waitlist***, and ***Mint***, respectively, **the difference in period affects the difference in time to access and fee for minting**.

Thus the period should follow the list below

1. **These periods should be in the proper order**
2. **The periods should not overlap with other periods**
3. **Within the period, the end time should be greater than the start time**

However, we found that the *setPeriods* function (L197 - 212 in the code snippet below) allows the *DEV_ROLE* to mistakenly set time without following the list above, which leads to creating an unfairness for the platform users.

For example, mistakenly set the ***Waitlist*** time before the ***Whitelist*** time.

CSKNFT.sol

```
197 function setPeriods(
198     MintType _mintType,
199     uint256 startDate,
200     uint256 _endTime
201 ) public onlyRole(DEV_ROLE) {
202     if (_mintType == MintType.Mint) {
203         mintDates.MINT_START_DATE = startDate;
204         mintDates.MINT_END_DATE = _endTime;
205     } else if (_mintType == MintType.Whitelist) {
206         mintDates.START_WHITELIST = startDate;
207         mintDates.END_WHITELIST = _endTime;
208     } else if (_mintType == MintType.Waitlist) {
209         mintDates.START_WAITLIST = startDate;
210         mintDates.END_WAITLIST = _endTime;
```

```
211         }
212     }
```

Listing 17.1 The *setPeriods* function that allows setting the improper period

Recommendations

Since the original design of the *setPeriods* function allows the *DEV_ROLE* to mistakenly set time (as described above), we recommend revising the *setPeriods* function to follow the criteria listed below to fix this issue.

1. The periods should be in the proper order
2. The periods should not overlap with another
3. Within each period, the end time should be greater than the start time

CSKNFT.sol

```
197 function setPeriods(
198     uint256 startWhitelistTime,
199     uint256 endWhitelistTime,
200     uint256 startWaitlistTime,
201     uint256 endWaitlistTime,
202     uint256 startMintTime,
203     uint256 endMintTime,
204 ) public onlyRole(TIMELOCK_DEV_ROLE) {
205     require(endWhitelistTime > startWhitelistTime, "invalid whitelist time");
206     require(endWaitlistTime > startWaitlistTime, "invalid waitlist time");
207     require(endMintTime > startMintTime, "invalid mint time");
208
209     require((endWhitelistTime < startWaitlistTime) && (endWaitlistTime <
    startMintTime), "invalid periods");
210
211     mintDates.START_WHITELIST = startWhitelistTime;
212     mintDates.END_WHITELIST = endWhitelistTime;
213
214     mintDates.START_WAITLIST = startWaitlistTime;
215     mintDates.END_WAITLIST = endWaitlistTime;
216
217     mintDates.MINT_START_DATE = startMintTime;
218     mintDates.MINT_END_DATE = endMintTime;
219 }
```

Listing 17.2 The improved *setPeriods* function that checks the proper periods

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 18	Overpayment When Minting And Updating NFT		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol* *contracts/CSKNFT.sol*		
Locations	*CSKGen.sol: L: 190 - 228 (*The *genToken* function*)* *CSKNFT.sol: L: 277 - 303 (*The *updateFlagStatus* function*)*		

Detailed Issue

The *genToken* and *updateFlagStatus* functions allow users to mint and update tokens, respectively. In order to perform these actions, users are required to pay *ethAmount* amount for the *genToken* function (L223 in the code snippet 18.1) and *PORTAL_PRICE* amount for the *updateFlagStatus* function (L293 in the code snippet 18.2).

However, upon conducting our analysis, we discovered that **there is a possibility of overpaying the required amount due to the current *require* statement implemented in the *genToken* and *updateFlagStatus* functions**.

As a result, users can accidentally overpay the required amount, which may lead to the loss of overpaid funds.

CSKGen.sol

```
190 function genToken(MintInfo calldata info) external payable nonReentrant {
191     uint256 ethAmount;
192     MintType mintType;
193     require(block.timestamp <= info.timestamp + 1 minutes, "Times out");
194     require(info.minter == msg.sender, "not minter");
195
196     //verify
197     address signer = _verify(info);
198     require(signer == signWallet, "not signed");
199
200     if (info.mintType == 0) {
201         require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
202         require(
203             wlLists[MintType.Whitelist][msg.sender] == true,
204             "not whitelist"
```

```
205             ); //wl
206             ethAmount = WHITELIST_PRICE;
207             mintType = MintType.Whitelist;
208         } else if (info.mintType == 1) {
209             require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
210             require(
211                 wlLists[MintType.Whitelist][msg.sender] == true ||
212                     wlLists[MintType.Waitlist][msg.sender] == true,
213                 "not waitlist"
214             );
215             ethAmount = WAITLIST_PRICE;
216             mintType = MintType.Waitlist;
217         } else if (info.mintType == 2) {
218             require(mintTotalCount[msg.sender].PbRound == 0, "Pb Minted");
219             ethAmount = MINT_PRICE;
220             mintType = MintType.Mint;
221         }
222 
223         require(msg.value >= ethAmount, "Eth not enough.");
224         (bool sent, ) = adminWallet.call{value: msg.value}("");
225         require(sent, "Failed to send Ether");
226 
227         loopGenToken(mintType, info.metadata);
228     }
```

Listing 18.1 The *genToken* function of the *CSKGen* contract

CSKNFT.sol

```
277 function updateFlagStatus(SignInfo calldata _info)
278     external
279     payable
280     nonReentrant
281 {
282     require(ownerOf(_info.tokenId) == msg.sender, "Not Owner.");
283     require(
284         tokenInfo[_info.tokenId].isAvailable != _info.status,
285         "Same status."
286     );
287 
288     //verify
289     address signer = _verify(_info);
290     require(signer == signWallet, "not signed");
291 
292     if (_info.status) {
293         require(msg.value >= PORTAL_PRICE, "Eth not enough.");
294 
295         (bool sent, ) = adminWallet.call{value: msg.value}("");
296         require(sent, "Failed send");
```

```
297            tokenInfo[_info.tokenId].metadata = _info.metadata;
298        }
299
300        tokenInfo[_info.tokenId].isAvailable = _info.status;
301
302        emit ChangeItemStatus(msg.sender, _info);
303    }
```

Listing 18.2 The *updateFlagStatus* function of the *CSKNFT* contract

Recommendations

We recommend making changes to the *require* statements in the *genToken* and *updateFlagStatus* functions. Specifically, we suggest changing the *require* statements to receive only the exact amount required by the functions.

By implementing these changes, we can ensure that the user pays the correct amount required for the functions to execute and prevent any overpayments.

CSKGen.sol

```
190    function genToken(MintInfo calldata info) external payable nonReentrant {
191        uint256 ethAmount;
192        MintType mintType;
193        require(block.timestamp <= info.timestamp + 1 minutes, "Times out");
194        require(info.minter == msg.sender, "not minter");
195
196        //verify
197        address signer = _verify(info);
198        require(signer == signWallet, "not signed");
199
200        if (info.mintType == 0) {
201            require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
202            require(
203                wlLists[MintType.Whitelist][msg.sender] == true,
204                "not whitelist"
205            ); //wl
206            ethAmount = WHITELIST_PRICE;
207            mintType = MintType.Whitelist;
208        } else if (info.mintType == 1) {
209            require(mintTotalCount[msg.sender].WlRound == 0, "Wl Minted");
210            require(
211                wlLists[MintType.Whitelist][msg.sender] == true ||
212                    wlLists[MintType.Waitlist][msg.sender] == true,
213                "not waitlist"
214            );
215            ethAmount = WAITLIST_PRICE;
```

```
216            mintType = MintType.Waitlist;
217        } else if (info.mintType == 2) {
218            require(mintTotalCount[msg.sender].PbRound == 0, "Pb Minted");
219            ethAmount = MINT_PRICE;
220            mintType = MintType.Mint;
221        }
222
223        require(msg.value == ethAmount, "Invalid Amount");
224        (bool sent, ) = adminWallet.call{value: msg.value}("");
225        require(sent, "Failed to send Ether");
226
227        loopGenToken(mintType, info.metadata);
228    }
```

Listing 18.3 The improved *genToken* function of the *CSKGen* contract

CSKNFT.sol

```
277    function updateFlagStatus(SignInfo calldata _info)
278        external
279        payable
280        nonReentrant
281    {
282        require(ownerOf(_info.tokenId) == msg.sender, "Not Owner.");
283        require(
284            tokenInfo[_info.tokenId].isAvailable != _info.status,
285            "Same status."
286        );
287
288        //verify
289        address signer = _verify(_info);
290        require(signer == signWallet, "not signed");
291
292        if (_info.status) {
293            require(msg.value == PORTAL_PRICE, "Invalid Amount");
294
295            (bool sent, ) = adminWallet.call{value: msg.value}("");
296            require(sent, "Failed send");
297            tokenInfo[_info.tokenId].metadata = _info.metadata;
298        }
299
300        tokenInfo[_info.tokenId].isAvailable = _info.status;
301
302        emit ChangeItemStatus(msg.sender, _info);
303    }
```

Listing 18.4 The improved *updateFlagStatus* function of the *CSKNFT* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team fixed this issue as per our suggestion.

No. 19	Lack Of Validating Input Parameters		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol* *contracts/CSKNFT.sol*		
Locations	*Several functions throughout multiple contracts*		

Detailed Issue

We found that the *CSKGen* and *CSKNFT* contracts have several crucial states that can be set by the setter functions. However, we noticed that these setter functions lack input parameter validation, which could potentially lead to issues with the contract's functionality.

- The ***CSKGen*** contract with no validating zero address functions
  - The *constructor* function
  - The *setSingWallet* function
  - The *setRandomWorker* function
- The ***CSKNFT*** contract with no validating zero address functions
  - The *initialize* function
  - The *setAdminWallet* function
- The ***CSKNFT*** contract with no validating the function parameter
  - The *setBaseURI* function

Recommendations

We recommend validating all input parameters for the setter functions in the *CSKGen* and *CSKNFT* contracts. This is crucial to prevent any unexpected behavior and ensure that the contracts would function as intended.

Please apply the validations to the following functions:

- The ***CSKGen*** contract with no validating zero address functions
  - The *constructor* function
  - The *setSingWallet* function
  - The *setRandomWorker* function

- The ***CSKNFT*** contract with no validating zero address functions
  - The *initialize* function
  - The *setAdminWallet* function

- The ***CSKNFT*** contract with no validating the function parameter
  - The *setBaseURI* function

The example below shows how to validate the zero address and empty string.

CSKGen.sol

```
103 function setSingWallet(address _newSignWallet)
104     public
105     onlyRole(DEV_ROLE)
106 {
107     require(_newSignWallet != address(0), "Invalid _newSignWallet address");
108     signWallet = _newSignWallet;
109 }
```

Listing 19.1 The example of *validating the zero address*

CSKNFT.sol

```
133 function setBaseURI(string memory _baseTokenURI)
134     public
135     onlyRole(DEFAULT_ADMIN_ROLE)
136 {
137     require(bytes(_baseTokenURI).length != 0, "Invalid _baseTokenURI");
138     baseTokenURI = _baseTokenURI;
139 }
```

Listing 19.2 The example of validating empty string for *setBaseURI* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team fixed this issue as per our suggestion.

No. 20	Compiler Is Not Locked To Specific Version		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol* *contracts/CSKNFT.sol*		
Locations	*CSKGen: L2* *CSKNFT: L2*		

Detailed Issue

We found that the smart contracts in this project should be deployed with the compiler version used in the development and testing process.

The compiler version that is not strictly locked via the *pragma* statement may make the contract incompatible against unforeseen circumstances.

List of smart contracts that should lock to the specific version.

- ***CSKGen*.sol**
- ***CSKNFT*.sol**

An example code that is not locked to a specific version (e.g., using => or ^ directive) is shown below.

CSKGen.sol

```
1 // SPDX-License-Identifier: CODESEKAI
2 pragma solidity ^0.8.0;
```

Listing 20.1 The *CSKGen* contract

Recommendations

We recommend locking the pragma version like the example code snippet below.

pragma solidity 0.8.19;
// or
pragma solidity =0.8.19;

```
contract SemVerFloatingPragmaFixed {

}
```

Reference: *https://swcregistry.io/docs/SWC-103*

Reassessment

The *CodeSekai* team locked the pragma version to v0.8.19.

No. 21	Compiler May Be Susceptible To Publicly Disclosed Bugs		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol* *contracts/CSKNFT.sol*		
Locations	*CSKGen: L2* *CSKNFT: L2*		

Detailed Issue

The *CSKGen* and *CSKNFT* smart contracts use an outdated Solidity compiler version (v0.8.0) which may be susceptible to publicly disclosed vulnerabilities. The latest compiler patch version is 0.8.19, which contains the list of known bugs as the following link:

https://docs.soliditylang.org/en/v0.8.19/bugs.html

The known bugs may not directly lead to the vulnerability, but it may increase an opportunity to trigger some attacks further.

An example smart contract that does not use the latest patch version is shown below.

CSKGen.sol

```
1  // SPDX-License-Identifier: CODESEKAI
2  pragma solidity ^0.8.0;
```

Listing 21.1 An example smart contract that does not use the latest patch version (v0.8.19)

Recommendations

We recommend using the latest patch version, v0.8.19, that fixes all known bugs.

Reassessment

The *CodeSekai* team fixed this issue by employing the patch version v0.8.19.

No. 22	Arbitrarily Setting NFT Minting Prices		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol*		
Locations	*CSKGen.sol L: 110 - 121 (The setPrice function)*		

Detailed Issue

We found that the *setPrice* function of the *CSKGen* contract allows for the arbitrary setting of the price values for minting NFTs. If the *DEV_ROLE* account sets the price values incorrectly, it could lead to an unfair price for the minting process.

Additionally, the current implementation of the function does not adhere to the fair price formula of ***MINT_PRICE > WAITLIST_PRICE >= WHITELIST_PRICE***, which could potentially give an unfair advantage to some minters and affect the overall fairness of the minting process.

CSKGen.sol

```
39  uint256 public MINT_PRICE = 0.11 ether;
40  uint256 public WHITELIST_PRICE = 0.08 ether;
41  uint256 public WAITLIST_PRICE = 0.08 ether;

    // (...SNIPPED...)

110 function setPrice(MintType _mintType, uint256 newPrice)
111     public
112     onlyRole(DEV_ROLE)
113 {
114     if (_mintType == MintType.Mint) {
115         MINT_PRICE = newPrice;
116     } else if (_mintType == MintType.Whitelist) {
117         WHITELIST_PRICE = newPrice;
118     } else if (_mintType == MintType.Waitlist) {
119         WAITLIST_PRICE = newPrice;
120     }
121 }
```

Listing 22.1 The *setPrice* function of the *CSKGen* contract

Recommendations

We recommend updating the *setPrice* function to enforce the fairness price formula of ***MINT_PRICE > WAITLIST_PRICE >= WHITELIST_PRICE*** and ensuring that the maximum allowed value for the mint price is not exceeded the ***MAXIMUM_MINT_PRICE*** constant as shown in the code snippet below.

This will ensure a fair price for each minting round and prevent any potential advantage for some minters.

CSKGen.sol

```
    // set max price properly
    uint256 public constant MAXIMUM_MINT_PRICE = 0.11 ether;

    // (...SNIPPED...)

110 function setPrice(MintType _mintType, uint256 newPrice)
112     public
113     onlyRole(DEV_ROLE)
114 {
115     require(newPrice <= MAXIMUM_MINT_PRICE, "New mint price exceeds the maximum
    allowed value");
116     if (_mintType == MintType.Mint) {
117         require(newPrice > WAITLIST_PRICE, "MINT_PRICE must be greater than
    WAITLIST_PRICE");
118         MINT_PRICE = newPrice;
119     } else if (_mintType == MintType.Whitelist) {
120         require(newPrice <= WAITLIST_PRICE, "WHITELIST_PRICE must be less than
    or equal to WAITLIST_PRICE");
121         WHITELIST_PRICE = newPrice;
122     } else if (_mintType == MintType.Waitlist) {
123         require(newPrice >= WHITELIST_PRICE && newPrice < MINT_PRICE,
    "WAITLIST_PRICE must be between WHITELIST_PRICE and MINT_PRICE");
124         WAITLIST_PRICE = newPrice;
125     }
126 }
```

Listing 22.2 The improved *setPrice* function of the *CSKGen* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team fixed this issue as per our suggestion.

No. 23	Potential Denial-Of-Service On The getUserTokenAndInfos Function		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKNFT.sol*		
Locations	*CSKNFT.sol L: 177 - 195*		

Detailed Issue

The *getUserTokenAndInfos* function (code snippet 23.1) is a getter function that facilitates retrieving a list of user assets owned by a specific member by iterating through the user's balance (L185 in the code snippet 23.1) to look up the token by the index.

The *CodeSekai* platform limits users to mint only two NFTs within the *Whitelist* or *Waitlist* round, and the *Public* round. However, the users could invoke the native *ERC721* functions (e.g. *transferFrom*, *safeTransferFrom*) to transfer NFT themselves directly.

As a result, users could end up owning more than two NFTs, which could potentially cause a denial-of-service issue when calling the *getUserTokenAndInfos* function.

Note that the getUserTokenAndInfos function is a getter function that might not consume gas for querying data (when querying data from an off-chain service). However, the underlying of the EVM (Ethereum Virtual Machine) node still counts in the gas being used by the call of the getUserTokenAndInfos function internally to prevent a denial-of-service attack on the EVM node itself. Therefore, the EVM node can reject the request if the querying process of the getUserTokenAndInfos function takes too much gas or takes too long to process.

CSKNFT.sol

```
177 function getUserTokenAndInfos(address userAddress)
178     public
179     view
180     returns (UserAsset[] memory)
181 {
182     uint256 balance = balanceOf(userAddress);
183     UserAsset[] memory userAssets = new UserAsset[](balance);
184 
185     for (uint32 i = 0; i < uint32(balance); i++) {
186         uint256 tokenId = tokenOfOwnerByIndex(userAddress, i);
187         string memory metadata = tokenInfo[tokenId].metadata;
188         bool status = checkFlagStatus(tokenId);
189 
190         userAssets[i].tokenId = tokenId;
191         userAssets[i].isAvailable = status;
192         userAssets[i].metadata = metadata;
193     }
194     return userAssets;
195 }
```

Listing 23.1 The *getUserTokenAndInfos* function that
is prone to the denial-of-service issue

Recommendations

One possible mitigating solution for this, **we recommend adding a new function (overloaded function) that applies the pagination concept.**

We provide the recommended code to address this issue as a suggested remediation concept only. The recommended code below should be used as a guideline to address this issue only. The *CodeSekai* team should adjust the recommended code properly according to the business design.

CSKNFT.sol

```
324 function getUserTokenAndInfos(address userAddress, uint256 cursor, uint256
    resultsPerPage)
325     public
326     view
327     returns (UserAsset[] memory userAssets, uint256 newCursor)
328 {
329     uint256 balances = balanceOf(userAddress);
330     require(cursor <= balances, "cursor is out of range");
331     require(resultsPerPage > 0, "resultsPerPage cannot be 0");
332 
333     uint256 length = resultsPerPage;
334     if (length > balances - cursor) {
```

```
335            length = balances - cursor;
336        }
337
338        userAssets = new UserAsset[](length);
339        for (uint256 i = 0; i < length; i++) {
340            uint256 tokenId = tokenOfOwnerByIndex(userAddress, cursor + i);
341            string memory metadata = tokenInfo[tokenId].metadata;
342            bool status = checkFlagStatus(tokenId);
343
344            userAssets[i].tokenId = tokenId;
345            userAssets[i].isAvailable = status;
346            userAssets[i].metadata = metadata;
347        }
348        return (userAssets, cursor + length);
349    }
```

Listing 23.2 The new *getUserTokenAndInfosFrom* function that applies the pagination concept

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team fixed this issue as per our suggestion.

<table>
<tr><td>No. 24</td><td colspan="3">Recommended Improving Transparency And Traceability Of Crucial Variables</td></tr>
<tr><td rowspan="2">Risk</td><td rowspan="2">Low</td><td>Likelihood</td><td>Low</td></tr>
<tr><td>Impact</td><td>Medium</td></tr>
<tr><td>Functionality is in use</td><td>In use</td><td>Status</td><td>Partially Fixed</td></tr>
<tr><td>Associated Files</td><td colspan="3">contracts/CSKGen.sol
contracts/CSKNFT.sol</td></tr>
<tr><td>Locations</td><td colspan="3">CSKGen.sol: L: 36, 37, 68, 69, 86, and 87
CSKNFT.sol: L: 29, 30, 32, 33, 37, 89, and 90</td></tr>
</table>

Detailed Issue

We found certain variables from various contracts are declared as ***private*** that could potentially result in a lack of transparency and traceability. The mentioned variables are listed below:

- The ***CSKGen*** contract
  - The *adminWallet* address
  - The *signWallet* address
  - The *SIGNING_DOMAIN* string
  - The *SIGNATURE_VERSION* string
  - The *wlLists* mapping
  - The *mintTotalCount* mapping

- The ***CSKNFT*** contract
  - The *adminWallet* address
  - The *signWallet* address
  - The *SIGNING_DOMAIN* string
  - The *SIGNATURE_VERSION* string
  - The *mintDates* struct
  - The *tokenInfo* mapping

Recommendations

We recommend changing the declaration of the associated variables to ***public***, in order to enhance transparency and traceability.

Reassessment

The *CodeSekai* team partially fixed this issue by enhancing the transparency and traceability of certain associated variables.

However, The ***SIGNING_DOMAIN*** and ***SIGNATURE_VERSION*** variables in the *CSKGen* and *CSKNFT* contracts have been kept private.

No. 25	Recommended Event Emissions For Transparency And Traceability		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol* *contracts/CSKNFT.sol*		
Locations	*Several functions throughout multiple contracts*		

Detailed Issue

We consider operations of the following state-changing functions important and require proper event emissions for improving transparency and traceability:

- The ***CSKGen*** contract
  - The *setRandomWorker* function
  - The *setSingWallet* function
  - The *setPrice* function
  - The *setWhitelists* function
  - The *genToken* function
- The ***CSKNFT*** contract
  - The *setBaseURI* function
  - The *setPeriods* function
  - The *setAdminWallet* function
  - The *setPortalPrice* function

Recommendations

We recommend emitting relevant events on the associated functions to improve transparency and traceability.

Reassessment

The *CodeSekai* team fixed this issue as per our suggestion.

No. 26	Lack Of Checking Availability Of The Token ID		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKNFT.sol*		
Locations	*CSKNFT.sol L: 140 - 142, and 144 - 150*		

Detailed Issue

The *checkFlagStatus* function (L140 - 142 in the code snippet below) allows retrieving the *isAvailable* property by a specific token ID.

However, we found that this function **lacks checking whether the token is minted before retrieving data, which can lead to incorrect results when the token ID is not minted.**

The same issue occurs in the *checkMetadata* function (L144 - 150 in the code snippet below) for retrieving the metadata.

CSKNFT.sol

```
140 function checkFlagStatus(uint256 tokenId) public view returns (bool) {
141     return tokenInfo[tokenId].isAvailable;
142 }
143
144 function checkMetadata(uint256 tokenId)
145     public
146     view
147     returns (string memory)
148 {
149     return tokenInfo[tokenId].metadata;
150 }
```

Listing 26.1 The *checkFlagStatus* and *checkMetadata* functions of the *CSKNFT* contract

Recommendations

We recommend invoking the *_requireMinted* function provided by the *ERC721* contract to ensure that the token ID is minted before retrieving data. This will prevent the return of incorrect results in case the token ID is not minted.

CSKNFT.sol

```
140 function checkFlagStatus(uint256 tokenId) public view returns (bool) {
141     _requireMinted(tokenId);
142     return tokenInfo[tokenId].isAvailable;
143 }
144
145 function checkMetadata(uint256 tokenId)
146     public
147     view
148     returns (string memory)
149 {
150     _requireMinted(tokenId);
151     return tokenInfo[tokenId].metadata;
152 }
```

Listing 26.2 The improved *checkFlagStatus* and *checkMetadata* functions of the *CSKNFT* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team fixed this issue as per our suggestion.

No. 27	Recommended Removing Unused Interfaces		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol*		
Locations	*CSKGen.sol L: 4, and 5*		

Detailed Issue

We found that the *CSKGen* contract contains the unused *IERC721* and *IERC20* interface. These unused interfaces could potentially cause confusion or misunderstandings among users or developers when attempting to maintain or modify the source code.

Moreover, unused interfaces can also increase the complexity of the codebase and lead to unnecessary computational overhead.

CSKGen.sol

```
4   import "@openzeppelin/contracts/token/ERC721/IERC721.sol";
5   import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

Listing 27.1 The unused interfaces of the *CSKGen* contract

Recommendations

We recommend removing interfaces code from the smart contracts as it can reduce the contract's complexity and also help to reduce confusion among users or developers when maintaining the source code.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 28	Recommended Removing Unused Code		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol*		
Locations	*CSKGen.sol* • *L: 11 -15 (*The *IRANDOMWORKER* interface*)* • *L: 35 (*The *IRANDOMWORKER* variable*)* • *L: 96 - 101 (*The *setRandomWorker* function*)*		

Detailed Issue

The *CSKGen* contract introduces the *IRANDOMWORKER* interface that relies on an external contract. However, it was observed that this external contract has not been utilized in the implementation of the *CSKGen* contract, despite having a setter function (The *setRandomWorker* function).

This unused external contract may lead to confusion or potential errors for future maintainers or developers of the source code.

CSKGen.sol

```
11  interface IRANDOMWORKER {
12      function getRandomNumber(uint256 tokenId, address _msgSender)
13          external
14          returns (string memory);
15  }

    // (...SNIPPED...)

33  contract CSKGen is EIP712, AccessControl, ReentrancyGuard {
34      INFTCORE public nftCore;
35      IRANDOMWORKER private iRandomWorker;

        // (...SNIPPED...)

96      function setRandomWorker(address _randomWokerAddr)
97          public
98          onlyRole(DEV_ROLE)
99      {
100         iRandomWorker = IRANDOMWORKER(_randomWokerAddr);
101     }
```

```
    // (...SNIPPED...)
229 }
```

Listing 28.1 The unused code of the *CSKGen* contract

Recommendations

We recommend removing unused code from the smart contracts as it can reduce the contract's complexity and also help to reduce confusion among users or developers when maintaining the source code.

Reassessment

The *CodeSekai* team adopted our recommended code to fix this issue.

No. 29	Recommended Removing Unused Imported Contract		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Partially Fixed
Associated Files	*contracts/CSKNFT.sol*		
Locations	*CSKNFT.sol L: 8 and 21*		

Detailed Issue

The *CSKNFT* contract inherits and imports the *Ownable* contract, but the functionality of the *Ownable* contract is not utilized in the *CSKNFT* contract.

The presence of this unused code might create confusion or misinterpretation among developers or users who are trying to maintain or make changes to the source code.

CSKNFT.sol

```
8   import "@openzeppelin/contracts/access/Ownable.sol";

    // (...SNIPPED...)

15  contract CSKNFT is
16      ERC721,
17      EIP712,
18      ERC721Enumerable,
19      ERC721Burnable,
20      AccessControl,
21      Ownable,
22      Initializable,
23      ReentrancyGuard
24  {
       // (...SNIPPED...)
322 }
```

Listing 29.1 The unused imported contract of the *CSKNFT* contract

Recommendations

We recommend removing unused code from the smart contracts as it can reduce the contract's complexity and also help to reduce confusion among users or developers when maintaining the source code.

Reassessment

The *CodeSekai* team partially fixed this issue by removing certain unused codes since it is not feasible to eliminate the use of an *Ownable* contract as *Opensea* relies on its functionality to import smart contracts to the market.

No. 30	Misspelling Of Crucial Function Name		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKGen.sol*		
Locations	*CSKGen.sol L: 103*		

Detailed Issue

The *setSingWallet* function is a crucial function that enables accounts with *DEV_ROLE* to set the new signer address. However, we found the misspelling function name of the *setSingWallet* function at line 103. This misspelling can lead to misunderstanding among users or developers when maintaining the source code.

CSKGen.sol

```
103 function setSingWallet(address _newSignWallet)
104     public
105     onlyRole(DEV_ROLE)
106 {
107     signWallet = _newSignWallet;
108 }
```

Listing 30.1 The misspelled function name of the *setSingWallet* function

Recommendations

We recommend revising the misspelled function name of the *setSingWallet* function.

CSKGen.sol

```
103 function setSignWallet(address _newSignWallet)
104     public
105     onlyRole(DEV_ROLE)
106 {
107     signWallet = _newSignWallet;
108 }
```

Listing 30.2 The improved function name

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team fixed this issue as per our suggestion.

No. 31	Recommended Adding Event Indexes		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CSKNFT.sol*		
Locations	*CSKNFT.sol L: 92 - 97, 99*		

Detailed Issue

We found that the *MintNft* and *ChangeItemStatus* events being emitted in the contract do not have the *indexed* modifier, **which results in a lack of traceability and makes it difficult to search for specific events in the event logs.**

CSKNFT.sol

```
92  event MintNft(
93      address userAddress,
94      uint256 tokenId,
95      uint256 createdAt,
96      MintType _mintType
97  );
98
99  event ChangeItemStatus(address userAddress, SignInfo);
```

Listing 31.1 The *MintNft* and *ChangeItemStatus* events

Recommendations

We recommend adding the *indexed* modifier to the *MintNft* and *ChangeItemStatus* events in the contract to improve traceability and facilitate more efficient searching of the event logs.

CSKNFT.sol

```
92  event MintNft(
93      address indexed userAddress,
94      uint256 indexed tokenId,
95      uint256 createdAt,
96      MintType indexed _mintType
97  );
```

```
98
99 event ChangeItemStatus(address indexed userAddress, SignInfo);
```

Listing 31.1 The improved *MintNft* and *ChangeItemStatus* events

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team fixed this issue as per our suggestion.

No. 32	Recommended Enforcing Checks-Effects-Interactions Pattern		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Partially Fixed
Associated Files	*contracts/CSKGen.sol* *contracts/CSKNFT.sol*		
Locations	*CSKGen.sol L: 152 - 165 (The loopGenToken function)* *CSKNFT.sol* • *L: 214 - 223 (The delMint function)* • *L: 277 - 303 (The updateFlagStatus function)*		

Detailed Issue

We noticed that the functions below ***do not follow the checks-effects-interactions pattern***, which is the best practice coding style to prevent potential reentrancy attacks.

List of functions that do not follow the checks-effects-interactions pattern.

- The ***loopGenToken*** function in the *CSKGen* contract
- The ***delMint*** function in the *CSKNFT* contract
- The ***updateFlagStatus*** function in the *CSKNFT* contract

Even if there are no reentrancy issues, we recommend that the list of functions above should enforce the *checks-effects-interactions* pattern.

For example, in the code snippet below, the *loopGenToken* function invokes the external *mint* function (*interactions part*) before updating the state variables (*effects part*).

CSKGen.sol

```
152 function loopGenToken(MintType _mintType, uint256 metadata) internal {
153     string memory results = Strings.toString(metadata);
154 
155     if (_mintType == MintType.Mint) {
156         nftCore.mint(msg.sender, results, INFTCORE.MintType.Mint);
157         mintTotalCount[msg.sender].PbRound += 1;
158     } else if (_mintType == MintType.Whitelist) {
159         nftCore.mint(msg.sender, results, INFTCORE.MintType.Whitelist);
```

```
160            mintTotalCount[msg.sender].WlRound += 1;
161        } else if (_mintType == MintType.Waitlist) {
162            nftCore.mint(msg.sender, results, INFTCORE.MintType.Waitlist);
163            mintTotalCount[msg.sender].WlRound += 1;
164        }
165    }
```

Listing 32.1 The *loopGenToken* function that
does not follow the *checks-effects-interactions* pattern

Recommendations

We recommend enforcing the checks-effects-interactions pattern to all of the functions below.

- The ***loopGenToken*** function in the *CSKGen* contract
- The ***delMint*** function in the *CSKNFT* contract
- The ***updateFlagStatus*** function in the *CSKNFT* contract

The example below is how to fix this issue, **we moved the *interactions* part (the *loopGenToken* function) to get executed after the *effects* part**.

CSKGen.sol

```
140 function loopGenToken(MintType _mintType, uint256 metadata) internal {
141     string memory results = Strings.toString(metadata);
142 
143     if (_mintType == MintType.Mint) {
144         mintTotalCount[msg.sender].PbRound += 1;
145         nftCore.mint(msg.sender, results, INFTCORE.MintType.Mint);
146     } else if (_mintType == MintType.Whitelist) {
147         mintTotalCount[msg.sender].WlRound += 1;
148         nftCore.mint(msg.sender, results, INFTCORE.MintType.Whitelist);
149     } else if (_mintType == MintType.Waitlist) {
150         mintTotalCount[msg.sender].WlRound += 1;
151         nftCore.mint(msg.sender, results, INFTCORE.MintType.Waitlist);
152     }
    }
```

Listing 32.2 The improved *loopGenToken* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *CodeSekai* team partially fixed this issue by enforcing the *checks-effects-interactions* pattern.

However, the ***delMint*** *and* ***updateFlagStatus*** functions still do not follow the *checks-effects-interactions* pattern.

Appendix

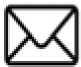
About Us

Founded in 2020, Valix Consulting is a blockchain and smart contract security firm offering a wide range of cybersecurity consulting services such as blockchain and smart contract security consulting, smart contract security review, and smart contract security audit.

Our team members are passionate cybersecurity professionals and researchers in the areas of private and public blockchain technology, smart contract, and decentralized application (DApp).

We provide a service for assessing and certifying the security of smart contracts. Our service also includes recommendations on smart contracts' security and gas optimization to bring the most benefit to users and platform creators.

Contact Information



info@valix.io

https://www.facebook.com/ValixConsulting

https://twitter.com/ValixConsulting

https://medium.com/valixconsulting

References

Title	Link
OWASP Risk Rating Methodology	https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
Smart Contract Weakness Classification and Test Cases	https://swcregistry.io/

© 2021 Valix Consulting Company Limited. All rights reserved.