Drakon+ NFT

Smart Contract Audit Report
Prepared for FIIT Token

FIIT TOKEN

Date Issued:	Jan 20, 2023
Project ID:	AUDIT2022057
Version:	v1.0
Confidentiality Level:	Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022057
Version	v1.0
Client	FIIT Token
Project	Drakon+ NFT
Auditor(s)	Fungkiat Phadejtaku Phitchakorn Apiratisakul
Author(s)	Phitchakorn Apiratisakul
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Jan 20, 2023	Full report	Phitchakorn Apiratisakul

Contact Information

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

Table of Contents

1. Executive Summary

As requested by FIIT Token, Inspex team conducted an audit to verify the security posture of the Drakon+ NFT smart contracts between Dec 14, 2022 and Dec 15, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Drakon+ NFT smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

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

1.2. Disclaimer

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

2. Project Overview

2.1. Project Introduction

Drakon+ NFT is the NFT collection from FiiT. The platform has Drakon+ NFT with a limited supply of 2000 and allows users to export Drakon+ NFT, point tokens from in-game assets into their wallet; the user can also import Drakon+ NFT, point tokens in their wallet to in-game assets.

Scope Information:

Project Name	Drakon+ NFT
Website	https://www.fiittoken.io/
Smart Contract Type	Ethereum Smart Contract
Chain	Polygon
Programming Language	Solidity
Category	NFT

Audit Information:

Audit Method	Whitebox
Audit Date	Dec 14, 2022 - Dec 15, 2022
Reassessment Date	Dec 28, 2022

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

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

2.2. Scope

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

Initial Audit: (Commit: 8129d23afed5198f2026f04d2cb5ef13d97b99f4)

Contract	Location (URL)
FiitTokenDrakonPlus	https://github.com/FiiT-Token/fiittoken-smartcontracts/blob/8129d23afe/contracts/FiitTokenDrakonPlus.sol
NftConverter	https://github.com/FiiT-Token/fiittoken-smartcontracts/blob/8129d23afe/contracts/NftConverter.sol
VerifySignature	https://github.com/FiiT-Token/fiittoken-smartcontracts/blob/8129d23afe/contracts/VerifySignature.sol

Reassessment: (Commit: 98331cdb8c69bb5c3a3f4f4a58749299ccd55e7b)

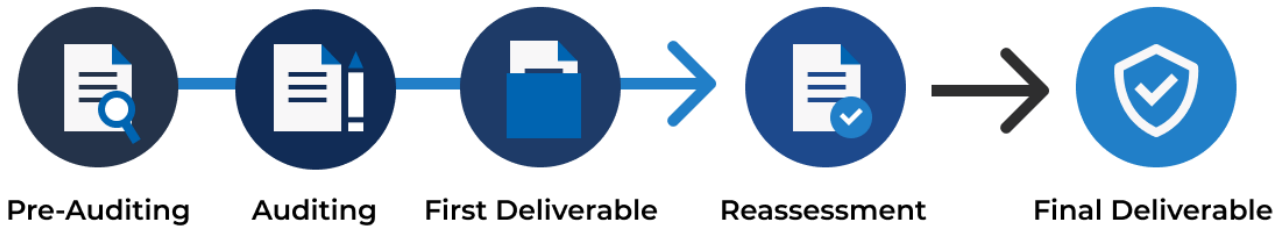
Contract	Location (URL)
FiitTokenDrakonPlus	https://github.com/FiiT-Token/fiittoken-smartcontracts/blob/98331cdb8c/contracts/FiitTokenDrakonPlus.sol
NftConverter	https://github.com/FiiT-Token/fiittoken-smartcontracts/blob/98331cdb8c/contracts/NftConverter.sol
VerifySignature	https://github.com/FiiT-Token/fiittoken-smartcontracts/blob/98331cdb8c/contracts/VerifySignature.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

3. Methodology

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

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



3.1. Test Categories

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

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

3.2. Audit Items

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

The following audit items were checked during the auditing activity:

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

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

3.3. Risk Rating

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

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

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

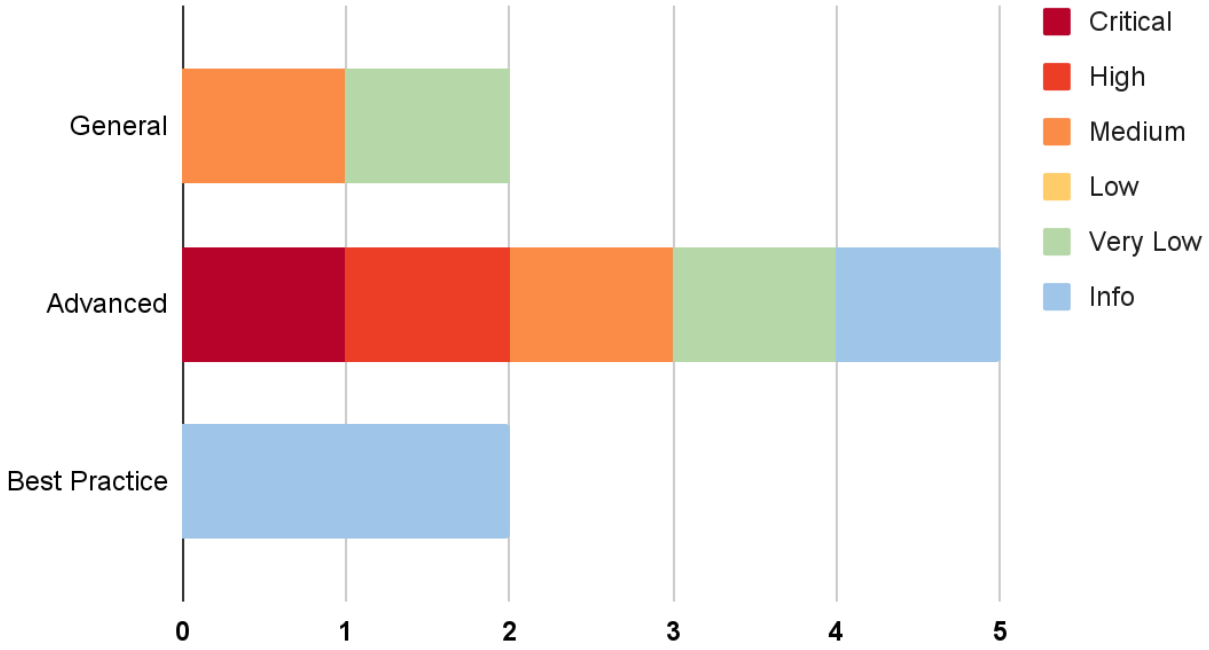
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would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

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

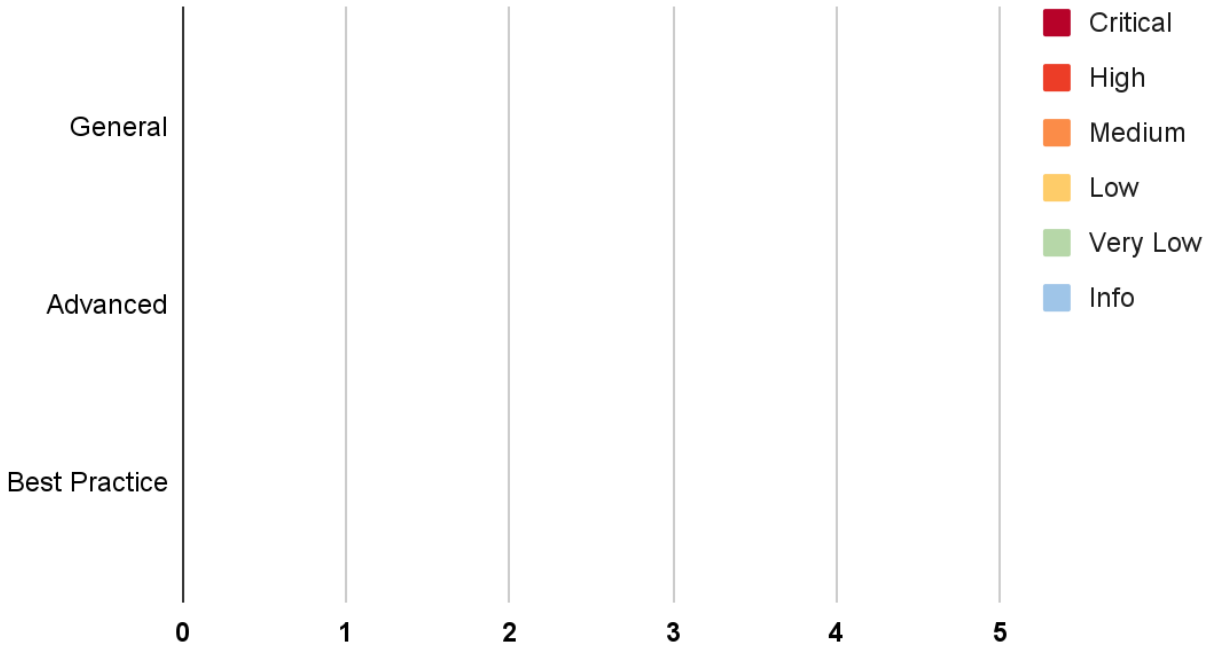
4. Summary of Findings

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

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

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

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

ID	Title	Category	Severity	Status
IDX-001	Arbitrary Update of Critical State Variables	Advanced	Critical	Resolved
IDX-002	Signature Reutilization	Advanced	High	Resolved
IDX-003	Centralized Control of State Variable	General	Medium	Resolved
IDX-004	Total Balance of NFT Exceeds Max Total Supply	Advanced	Medium	Resolved
IDX-005	Incorrect MAX_SUPPLY Validation in reserveNFTs() Function	Advanced	Very Low	Resolved
IDX-006	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-007	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved
IDX-008	Improper Function Visibility	Best Practice	Info	Resolved
IDX-009	Unintentional Design Left in Contract	Advanced	Info	Resolved

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

5. Detailed Findings Information

5.1. Arbitrary Update of Critical State Variables

ID	IDX-001
Target	NftConverter
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: Critical** **Impact: High** The attacker can control the `nftAddress` and `nftOwnerAddress` state variables, which can be used to impersonate themselves as the platform's NFT wallet, and also deny the service of the importing and exporting functions. **Likelihood: High** Anyone can execute the `setNftAddress()` and `setNftOwnerAddress()` functions to initiate the attack.
Status	Resolved The FIIT Token team has fixed this issue by removing the functions `setNftAddress()` and `setNftOwnerAddress()` in commit `98331cdb8c69bb5c3a3f4f4a58749299ccd55e7b`.

5.1.1. Description

The `NftConverter` contract allows anyone to export an NFT from off-chain to on-chain by executing the `exportNft()` function, or conversely, import an NFT from on-chain to off-chain by executing the `importNft()` function.

Technically, those two functions are working with the NFT contract, which is defined in the `nftAddress` state variable. If the user has provided correct data and a valid signature, the `exportNft()` function will transfer the target NFT from the `nftOwnerAddress` to the `msg.sender`, or the `importNft()` function will transfer the target NFT from `msg.sender` to the `nftOwnerAddress`.

NftConverter.sol

```
42  function exportNft(uint tokenId, uint256 nonce, uint256 expiredAt, bytes memory
    signature) public {
43      bytes32 messageHash = getMessageHash(
44          msg.sender,
45          tokenId, // token id
46          "export",
47          nonce,
48          expiredAt
```

```
49      );
50
51      require(recoverSigner(messageHash, signature) == nftOwnerAddress, "Export:
        Invalid signature");
52
53      require(expiredAt >= block.timestamp, "Export: Transaction Expired");
54
55      IERC721A(nftAddress).safeTransferFrom(nftOwnerAddress, msg.sender, tokenId,
        "");
56  }
57
58  function importNft(uint tokenId, uint256 nonce, uint256 expiredAt, bytes memory
    signature) public {
59      bytes32 messageHash = getMessageHash(
60          msg.sender,
61          tokenId, // token id
62          "import",
63          nonce,
64          expiredAt
65      );
66
67      require(recoverSigner(messageHash, signature) == msg.sender, "Export:
        Invalid signature");
68
69      require(expiredAt >= block.timestamp, "Import: Transaction Expired");
70
71      IERC721A(nftAddress).safeTransferFrom(msg.sender, nftOwnerAddress, tokenId,
        "");
72  }
```

However, in the current implementation, anyone can update the `nftOwnerAddress` state variable, which is the platform's NFT holding wallet, and the `nftAddress` state variable which is the target NFT contract.

NftConverter.sol

```
74  function setNftAddress(address _nftAddress) public {
75      nftAddress = _nftAddress;
76  }
```

NftConverter.sol

```
83  function setNftOwnerAddress(address _nftOwnerAddress) public {
84      nftOwnerAddress = _nftOwnerAddress;
85  }
```

As a result, these failures could lead to the following scenarios:

Scenario 1: Impersonate their NFT wallet as the platform's NFT wallet

The attacker only needs to execute the `setNftOwnerAddress()` function to update the `nftOwnerAddress` state variable to their NFT wallet. Hereafter, if any user executes the `importNft()` function to import an NFT to the off-chain server, the contract will transfer that NFT from the user's NFT wallet to the attacker's NFT wallet, as shown in line 73.

NftConverter.sol

```
60 function importNft(uint tokenId, uint256 nonce, uint256 expiredAt, bytes memory
   signature) public {
61     bytes32 messageHash = getMessageHash(
62         msg.sender,
63         tokenId, // token id
64         "import",
65         nonce,
66         expiredAt
67     );
68
69     require(recoverSigner(messageHash, signature) == msg.sender, "Export:
   Invalid signature");
70
71     require(expiredAt >= block.timestamp, "Import: Transaction Expired");
72
73     IERC721A(nftAddress).safeTransferFrom(msg.sender, nftOwnerAddress, tokenId,
   "");
74 }
```

Scenario 2: Denial of Service

The attack has the ability to disable the `importNft()` and `exportNft()` functions by executing the `setNftAddress()` function to update `nftAddress` value to 0. For example, the user will not be able to export NFT from the platform because of line 51.

NftConverter.sol

```
42 function exportNft(uint tokenId, uint256 nonce, uint256 expiredAt, bytes memory
   signature) public {
43     bytes32 messageHash = getMessageHash(
44         msg.sender,
45         tokenId, // token id
46         "export",
47         nonce,
48         expiredAt
49     );
50
51     require(recoverSigner(messageHash, signature) == nftOwnerAddress, "Export:
   Invalid signature");
52
53     require(expiredAt >= block.timestamp, "Export: Transaction Expired");
```

```
54
55      IERC721A(nftAddress).safeTransferFrom(nftOwnerAddress, msg.sender, tokenId,
        "");
56  }
```

5.1.2. Remediation

Inspex suggests removing the ability to update the `nftAddress` and `nftOwnerAddress` state variables. This can be done by removing the `setNftAddress()` and `setNftOwnerAddress()` functions and only updating these state variables while initiating the contract, as shown in the following source code:

NftConverter.sol

```
18  constructor(address _nftAddress, address _nftOwnerAddress) {
19      require(_nftAddress != address(0), "The _nftAddress should not be the zero
        address");
20      require(_nftOwnerAddress != address(0), "The _nftOwnerAddress should not be
        the zero address");
21      nftAddress = _nftAddress;
22      nftOwnerAddress = _nftOwnerAddress;
23  }
```

If removing the functions is not possible, Inspex suggests mitigating the risk of this issue by applying the `onlyOwner` modifier to the functions and also emitting events, as shown in the following source code:

NftConverter.sol

```
73  event SetNftAddress(address _nftAddress);
74  function setNftAddress(address _nftAddress) public onlyOwner {
75      nftAddress = _nftAddress;
76      event SetNftAddress(_nftAddress);
77  }
```

NftConverter.sol

```
82  event SetNftOwnerAddress(address _nftOwnerAddress);
83  function setNftOwnerAddress(address _nftOwnerAddress) public onlyOwner {
84      nftOwnerAddress = _nftOwnerAddress;
85      event SetNftAddress(_nftAddress);
86  }
```

Inspex also suggests using the timelock mechanism to delay the changes for a reasonable amount of time at least 24 hours, if the mitigation solution is chosen.

5.2. Signature Reutilization

ID	IDX-002
Target	NftConverter VerifySignature
Category	Advanced Smart Contract Vulnerability
CWE	CWE-347: Improper Verification of Cryptographic Signature
Risk	**Severity: High** **Impact: High** The signature can be reused, resulting in the user being able to drain NFTs from the platform. **Likelihood: Medium** Profits from this issue can only be acquired by registered users.
Status	**Resolved** The FIIT Token team has fixed this issue by implementing a restriction condition as suggested in commit `14b3d0caa9385b2d2c572ba3257524c3c2aaf979`.

5.2.1. Description

The `NftConverter` and `VerifySignature` contracts allow users to freely interact with the claiming functions, where they can claim rewards or assets from the contract. Additionally, the functions are programmatically trusted in the signature verification process. If the user can provide correct data and a valid signature, then the user can claim what they require.

However, the contracts also allow a valid signature to be used more than once, and this could lead to the following scenarios:

Scenario 1: Stealing an NFT

In the `NftConverter` contract, the `exportNft()` and `importNft()` functions do not have a restriction condition to prevent the user from using a valid signature more than once.

NftConverter.sol

```
42  function exportNft(uint tokenId, uint256 nonce, uint256 expiredAt, bytes memory
    signature) public {
43      bytes32 messageHash = getMessageHash(
44          msg.sender,
45          tokenId, // token id
46          "export",
47          nonce,
```

```
48          expiredAt
49      );
50
51      require(recoverSigner(messageHash, signature) == nftOwnerAddress, "Export:
    Invalid signature");
52
53      require(expiredAt >= block.timestamp, "Export: Transaction Expired");
54
55      IERC721A(nftAddress).safeTransferFrom(nftOwnerAddress, msg.sender, tokenId,
    "");
56  }
57
58  function importNft(uint tokenId, uint256 nonce, uint256 expiredAt, bytes memory
    signature) public {
59      bytes32 messageHash = getMessageHash(
60          msg.sender,
61          tokenId, // token id
62          "import",
63          nonce,
64          expiredAt
65      );
66
67      require(recoverSigner(messageHash, signature) == msg.sender, "Export:
    Invalid signature");
68
69      require(expiredAt >= block.timestamp, "Import: Transaction Expired");
70
71      IERC721A(nftAddress).safeTransferFrom(msg.sender, nftOwnerAddress, tokenId,
    "");
72  }
```

Therefore, the attacker can steal an NFT by conducting the following steps:

1. Request a valid signature to export an NFT from the off-chain server, then execute the `exportNFT()` function to transfer the NFT and also keep the signature for step 4.
2. Sell the NFT in a marketplace.
3. Wait until the buyer imports the NFT to the off-chain server with the `importNft()` function.
4. Execute the `exportNFT()` function with the same signature as step 1 to transfer the NFT to the attacker's NFT wallet.

As a result, the attacker can use a signature more than once to steal the NFT from other players.

Scenario 2: Draining Points via the VerifySignature Contract

In the `VerifySignature` contract, the `verifyAndMint()` and `verifyAndBurn()` functions do not have a restriction condition to prevent the user from using a valid signature more than once.

VerifySignature.sol

```
96  function verifyAndMint(
97      address to,
98      address token,
99      uint256 amount,
100     uint256 nonce,
101     bytes memory signature
102 ) external {
103     // Check signature is sign by authorize signer
104     bool isValid = _verify(to,token,amount,nonce,signature);
105
106     require(isValid, "Verify: Not authorize address");
107
108     if (isValid) {
109         IBEP20(token).redeem(to, amount);
110     }
111 }
112
113 function verifyAndBurn(
114     address to,
115     address token,
116     uint256 amount,
117     uint256 nonce,
118     bytes memory signature
119 ) external {
120     bool isValid = _verify(to,token,amount,nonce,signature);
121     require(isValid, "Verify: Not authorize address");
122
123     if (isValid) {
124         IBEP20(token).convertToPoint(to, amount);
125     }
126 }
```

So, the attacker can drain points from the point token contract by conducting the following steps:

Assuming that the attacker has some points on the off-chain server.

1. Request a valid signature from the off-chain server for executing the `verifyAndMint()` function.
2. Repeatedly execute the `verifyAndMint()` to mint tokens to the attacker's wallet with a valid signature from step 1.
3. Execute the `verifyAndBurn()` function to convert the point token to offchain's point, then use it to take profits in the off-chain server.

As a result, the attacker's account on the off-chain server has significantly high points.

Scenario 3: Replay attacking in the VerifySignature Contract

In the `VerifySignature` contract, the `verifyAndBurn()` function does not have a restriction condition to prevent the user from using a valid signature more than once.

VerifySignature.sol

```
113 function verifyAndBurn(
114     address to,
115     address token,
116     uint256 amount,
117     uint256 nonce,
118     bytes memory signature
119 ) external {
120     bool isValid = _verify(to,token,amount,nonce,signature);
121     require(isValid, "Verify: Not authorize address");
122 
123     if (isValid) {
124         IBEP20(token).convertToPoint(to, amount);
125     }
126 }
```

As a result, after users execute the `verifyAndBurn()` function, the data in the transactions will be publicly accessible in the block explorer. Hence, the attacker now has the ability to agitate them by executing the `verifyAndBurn()` function with the same data as they did before. resulting in the point token in their wallet being converted against their will.

5.2.2. Remediation

Inspex suggests implementing a restriction condition to prevent a valid signature from being used more than once, as shown in the following source code:

For the `NftConverter` contract, Inspex suggests implementing a restriction condition to the `exportNft()` and `importNft()` functions, as shown in lines 41, 43-44 and 61-63.

NftConverter.sol

```
41 mapping(bytes => bool) private _signatureUseds;
42 function exportNft(uint tokenId, uint256 nonce, uint256 expiredAt, bytes memory
   signature) public {
43     require(_signatureUseds[signature] == false, "Export: Signature is already
   used");
44     _signatureUseds[signature] = true;
45     bytes32 messageHash = getMessageHash(
46         msg.sender,
47         tokenId, // token id
48         "export",
49         nonce,
50         expiredAt
51     );
```

```
52
53      require(recoverSigner(messageHash, signature) == nftOwnerAddress, "Export:
    Invalid signature");
54
55      require(expiredAt >= block.timestamp, "Export: Transaction Expired");
56
57      IERC721A(nftAddress).safeTransferFrom(nftOwnerAddress, msg.sender, tokenId,
    "");
58  }
59
60  function importNft(uint tokenId, uint256 nonce, uint256 expiredAt, bytes memory
    signature) public {
61      require(_signatureUseds[signature] == false, "Import: Signature is already
    used");
62      _signatureUseds[signature] = true;
63      bytes32 messageHash = getMessageHash(
64          msg.sender,
65          tokenId, // token id
66          "import",
67          nonce,
68          expiredAt
69      );
70
71      require(recoverSigner(messageHash, signature) == msg.sender, "Export:
    Invalid signature");
72
73      require(expiredAt >= block.timestamp, "Import: Transaction Expired");
74
75      IERC721A(nftAddress).safeTransferFrom(msg.sender, nftOwnerAddress, tokenId,
    "");
76  }
```

For the `VerifySignature` contract, Inspex suggests implementing a restriction condition to the `verifyAndMint()` and `verifyAndBurn()` functions, as shown in lines 103-104 and 122-123.

VerifySignature.sol

```
95  mapping(bytes => bool) private _signatureUseds;
96  function verifyAndMint(
97      address to,
98      address token,
99      uint256 amount,
100     uint256 nonce,
101     bytes memory signature
102 ) external {
103     require(_signatureUseds[signature] == false, "Verify: Signature is already
    used");
104     _signatureUseds[signature] = true;
```

```
105     // Check signature is sign by authorize signer
106     bool isValid = _verify(to,token,amount,nonce,signature);
107 
108     require(isValid, "Verify: Not authorize address");
109 
110     if (isValid) {
111         IBEP20(token).redeem(to, amount);
112     }
113 }
114 
115 function verifyAndBurn(
116     address to,
117     address token,
118     uint256 amount,
119     uint256 nonce,
120     bytes memory signature
121 ) external {
122     require(_signatureUseds[signature] == false, "Verify: Signature is already
    used");
123     _signatureUseds[signature] = true;
124     bool isValid = _verify(to,token,amount,nonce,signature);
125     require(isValid, "Verify: Not authorize address");
126 
127     if (isValid) {
128         IBEP20(token).convertToPoint(to, amount);
129     }
130 }
```

5.3. Centralized Control of State Variable

ID	IDX-003
Target	FiitTokenDrakonPlus
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	Resolved The FIIT Token team has fixed this issue by removing the functions `reserveNFTs()` and `setBaseURI()` in commit `98331cdb8c69bb5c3a3f4f4a58749299ccd55e7b`.

5.3.1. Description

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

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

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
FiitTokenDrakonPlus.sol (L:25)	FiitTokenDrakonPlus	reserveNFTs()	onlyOwner
FiitTokenDrakonPlus.sol (L:37)	FiitTokenDrakonPlus	setBaseURI()	onlyOwner

5.3.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests implementing a community-run smart contract governance to control the use of these functions. If removing the functions or implementing the smart contract governance is not possible, Inspex suggests mitigating the risk of this issue by using a timelock mechanism to delay the changes for a reasonable amount of time at least 24 hours.

5.4. Total Balance of NFT Exceeds Max Total Supply

ID	IDX-004
Target	FiitTokenDrakonPlus
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium When the functions that are using `_safeMint()` function were called, the `_tokenIds` state did not increase correctly, resulting in the NFT being minted exceeding `MAX_SUPPLY`. **Likelihood:** Medium It is not likely that anyone will call the `mintNFTs()` function to buy a no-value NFT with 0.01 ether then mint the NFT and only the owner can use the `reserveNFTs()` function to mint NFTs.
Status	Resolved The FIIT Token team has fixed this issue by minting an NFT in the maximum supply in constructor instead of minting in the function `reserveNFTs()` and removing the functions `reserveNFTs()` and `mintNFTs()` in commit `98331cdb8c69bb5c3a3f4f4a58749299ccd55e7b` .

5.4.1. Description

In the `FiitTokenDrakonPlus` contract, the functions `reserveNFTs()` and `mintNFTs()` are used to mint an NFT, and these functions validate an input amount that should not be more than the `MAX_SUPPLY` in lines 28 and 44.

FiitTokenDrakonPlus.sol

```
25 function reserveNFTs(uint256 amount) public onlyOwner {
26     uint totalMinted = _tokenIds.current();
27
28     require(totalMinted.add(amount) < MAX_SUPPLY, "Not enough NFTs left to
   reserve");
29
30     _safeMint(msg.sender, amount);
31 }
```

FiitTokenDrakonPlus.sol

```
41 function mintNFTs(uint _count) public payable {
42     uint totalMinted = _tokenIds.current();
```

```
43
44      require(totalMinted.add(_count) <= MAX_SUPPLY, "Not enough NFTs left!");
45      require(_count >0 && _count <= MAX_PER_MINT, "Cannot mint specified number
        of NFTs.");
46      require(msg.value >= PRICE.mul(_count), "Not enough ether to purchase
        NFTs.");
47
48      _safeMint(msg.sender, _count);
49  }
```

However, the `_tokenIds` state did not increase after the NFT was minted, causing the `totalMinted` state to be impossible to equal the current supply and always pass the `MAX_SUPPLY` validation.

5.4.2. Remediation

Inspex suggests implementing the total supply validation properly by using `totalSupply()` instead.

FiitTokenDrakonPlus.sol

```
25  function reserveNFTs(uint256 amount) public onlyOwner {
26      require(totalSupply() + amount <= MAX_SUPPLY, "Not enough NFTs left to
        reserve");
27
28      _safeMint(msg.sender, amount);
29  }
```

FiitTokenDrakonPlus.sol

```
41  function mintNFTs(uint _count) public payable {
42      require(totalSupply() + _count  <= MAX_SUPPLY, "Not enough NFTs left!");
43      require(_count >0 && _count <= MAX_PER_MINT, "Cannot mint specified number
        of NFTs.");
44      require(msg.value >= PRICE.mul(_count), "Not enough ether to purchase
        NFTs.");
45
46      _safeMint(msg.sender, _count);
47  }
```

5.5. Incorrect MAX_SUPPLY Validation in reserveNFTs() Function

ID	IDX-005
Target	FiitTokenDrakonPlus
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low The owner cannot use `reserveNFTs()` function to mint the NFT as an amount of input. **Likelihood:** Low Only the owner can use the `reserveNFTs()` function and the input amount needs to be equal to `MAX_SUPPLY`.
Status	Resolved The FIIT Token team has fixed this issue by removing the function `reserveNFTs()` in commit `98331cdb8c69bb5c3a3f4f4a58749299ccd55e7b`.

5.5.1. Description

In the contract `FiitTokenDrakonPlus`, The owner can use `reserveNFTs()` function to mint NFTs for a specific amount to their wallet. But, if the owner wants to mint with the amount that the sum of `totalMinted` and `amount` equal to `MAX_SUPPLY`, the function will not be able to mint NFTs due to the comparison in line 28.

FiitTokenDrakonPlus.sol

```
25 function reserveNFTs(uint256 amount) public onlyOwner {
26     uint totalMinted = _tokenIds.current();
27
28     require(totalMinted.add(amount) < MAX_SUPPLY, "Not enough NFTs left to
   reserve");
29
30     _safeMint(msg.sender, amount);
31 }
```

5.5.2. Remediation

Inspex suggests using "less than or equal to" operator to validate the amount with `MAX_SUPPLY`.

FiitTokenDrakonPlus.sol

```
25 function reserveNFTs(uint256 amount) public onlyOwner {
26     uint totalMinted = _tokenIds.current();
```

```
27
28      require(totalMinted.add(amount) <= MAX_SUPPLY, "Not enough NFTs left to
        reserve");
29
30      _safeMint(msg.sender, amount);
31  }
```

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

5.6. Insufficient Logging for Privileged Functions

ID	IDX-006
Target	FiitTokenDrakonPlus
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved The FIIT Token team has fixed this issue by adding an event to the withdraw function as suggested in commit `e4f9dfe54c23f09dad608fedc9f2fb0e51ac2526`.

5.6.1. Description

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

For example, the owner can withdraw the ether in the contract by executing the `withdraw()` function in the `FiitTokenDrakonPlus` contract, and no events are emitted.

FiitTokenDrakonPlus.sol

```
51 function withdraw() public payable onlyOwner {
52     uint balance = address(this).balance;
53     require(balance > 0, "No ether left to withdraw");
54
55     (bool success, ) = (msg.sender).call{value: balance}("");
56     require(success, "Transfer failed.");
57 }
```

The privileged functions without sufficient logging are as follows:

File	Contract	Function
FiitTokenDrakonPlus.sol (L: 37)	FiitTokenDrakonPlus	setBaseURI()
FiitTokenDrakonPlus.sol (L: 51)	FiitTokenDrakonPlus	withdraw()

5.6.2. Remediation

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

FiitTokenDrakonPlus.sol

```
51  event Withdraw(address owner, uint256 amount);
52  function withdraw() public payable onlyOwner {
53      uint balance = address(this).balance;
54      require(balance > 0, "No ether left to withdraw");
55
56      (bool success, ) = (msg.sender).call{value: balance}("");
57      require(success, "Transfer failed.");
58      emit Withdraw(msg.sender, balance);
59  }
```

5.7. Inexplicit Solidity Compiler Version

ID	IDX-007
Target	FiitTokenDrakonPlus NftConverter VerifySignature
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The FIIT Token team has fixed this issue by changing the solidity version to the explicit latest solidity version as suggested in commit `e4f9dfe54c23f09dad608fedc9f2fb0e51ac2526`.

5.7.1. Description

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

FiitTokenDrakonPlus.sol

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

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

File	Version
FiitTokenDrakonPlus (L:2)	^0.8.0
NftConverter (L:2)	^0.8.0
VerifySignature (L:3)	^0.8.3

5.7.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.8 is v0.8.17.

(https://github.com/ethereum/solidity/releases)

FiitTokenDrakonPlus.sol

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

5.8. Improper Function Visibility

ID	IDX-008
Target	FiitTokenDrakonPlus NftConverter
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The FIIT Token team has fixed this issue by removing the function `reserveNFTs()` in commit `98331cdb8c69bb5c3a3f4f4a58749299ccd55e7b`.

5.8.1. Description

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

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

FiitTokenDrakonPlus.sol

```
25 function reserveNFTs(uint256 amount) public onlyOwner {
26     uint totalMinted = _tokenIds.current();
27
28     require(totalMinted.add(amount) < MAX_SUPPLY, "Not enough NFTs left to
   reserve");
29
30     _safeMint(msg.sender, amount);
31 }
```

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

File	Contract	Function
FiitTokenDrakonPlus.sol (L: 25)	FiitTokenDrakonPlus	reserveNFTs()

FiitTokenDrakonPlus.sol (L: 37)	FiitTokenDrakonPlus	setBaseURI()
FiitTokenDrakonPlus.sol (L: 41)	FiitTokenDrakonPlus	mintNFTs()
FiitTokenDrakonPlus.sol (L: 51)	FiitTokenDrakonPlus	withdraw()
NftConverter.sol (L: 42)	NftConverter	exportNft()
NftConverter.sol (L: 58)	NftConverter	importNft()
NftConverter.sol (L: 74)	NftConverter	setNftAddress()
NftConverter.sol (L: 78)	NftConverter	getNftAddress()
NftConverter.sol (L: 83)	NftConverter	setNftOwnerAddress()
NftConverter.sol (L:87)	NftConverter	getNftOwnerAddress()

5.8.2. Remediation

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

FiitTokenDrakonPlus.sol

```
25 function reserveNFTs(uint256 amount) external onlyOwner {
26     uint totalMinted = _tokenIds.current();
27
28     require(totalMinted.add(amount) <= MAX_SUPPLY, "Not enough NFTs left to
   reserve");
29
30     _safeMint(msg.sender, amount);
31 }
```

5.9. Unintentional Design Left in Contract

ID	IDX-009
Target	FiitTokenDrakonPlus
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The FIIT Token team has fixed this issue by changing the `MAX_SUPPLY` to 2000 and removing the `mintNFTs()` function in commit `98331cdb8c69bb5c3a3f4f4a58749299ccd55e7b`.

5.9.1. Description

As mentioned in the public document (FiiTToken-WhitePaper.pdf), in the `FiitTokenDrakonPlus` contract, the total supply should be 2000, and the `mintNFTs()` function should not be allowed to be used by users anymore.

Since anyone can use `mintNFTs()` function to mint NFTs before other users or mint NFTs to increase the `totalSupply` to reach the maximum supply. If the platform wants to increase the NFT supply, the `tokenId` will be out of order as expected.

VerifySignature.sol

```
17 uint public constant MAX_SUPPLY = 10000;
```

VerifySignature.sol

```
41 function mintNFTs(uint _count) public payable {
42     uint totalMinted = _tokenIds.current();
43
44     require(totalMinted.add(_count) <= MAX_SUPPLY, "Not enough NFTs left!");
45     require(_count >0 && _count <= MAX_PER_MINT, "Cannot mint specified number
   of NFTs.");
46     require(msg.value >= PRICE.mul(_count), "Not enough ether to purchase
   NFTs.");
47
48     _safeMint(msg.sender, _count);
49 }
```

5.9.2. Remediation

Inspex suggests changing the `MAX_SUPPLY` value to 2000 according to the document, as shown in the following source code:

VerifySignature.sol

```
17  uint public constant MAX_SUPPLY = 2000;
```

For the `mintNFTs()` function, Inspex suggests removing this function to prevent unintentionally minting that will increase the `tokenId` value, which is unexpected for business logic.

6. Appendix

6.1. About Inspex

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

Follow Us On:

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

inspeX

CYBERSECURITY PROFESSIONAL SERVICE