HAECHI AUDIT

Luxon - airdrop/gacha

Smart Contract Security Analysis

Published on : Sept 19, 2022

Version v1.1

HAECHI AUDIT

Smart Contract Audit Certificate

Luxon - airdrop/gacha

Security Report Published by HAECHI AUDIT

v1.1 Sept 19, 2022

Auditor : Andy Koo, Jade Han

Findings

Severity of Issues	Findings	Resolved	Unresolved	Acknowledged	Comment
Critical	-	-	-	-	-
Major	1	1	-	-	-
Minor	1	1	-	-	-
Tips	4	4	-	-	-

TABLE OF CONTENTS

6 Issues (0 Critical, 1 Major, 1 Minor, 4 Tips) Found, all issues were resolved.

ABOUT US

HAECHI AUDIT believes in the power of cryptocurrency and the next paradigm it will bring. We have the vision to empower the next generation of finance. By providing security and trust in the blockchain industry, we dream of a world where everyone has easy access to blockchain technology.

HAECHI AUDIT is a flagship service of HAECHI LABS, the leader of the global blockchain industry. HAECHI AUDIT provides specialized and professional smart contract security auditing and development services.

We are a team of experts with years of experience in the blockchain field and have been trusted by 400+ project groups. Our notable partners include Sushiswap,1inch, Klaytn, Badger, etc.

HAECHI AUDIT is the only blockchain technology company selected for the Samsung Electronics Startup Incubation Program in recognition of our expertise. We have also received technology grants from the Ethereum Foundation and Ethereum Community Fund.

Inquiries: audit@haechi.io
Website: audit.haechi.io

INTRODUCTION

This report was prepared to audit the security of the Airdrop and Gacha system-related contracts developed by the Luxon team. HAECHI AUDIT conducted the audit focusing on whether the system created by the Luxon team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the Airdrop and Gacha implementation.

In detail, we have focused on the following

- Unintended behavior on the process of Airdrop.
- Possibility of exploiting and abusing predictability of the Gacha system.
- Project availability issues like Denial of Service.
- Storage variable access control.
- Adequate implementation of ERC721 and ERC1155 spec.
- Existence of known smart contract vulnerabilities.

CRITICAL	Critical issues must be resolved as critical flaws that can harm a wide range of users.
MAJOR	Major issues require correction because they either have security problems or are implemented not as intended.
MINOR	Minor issues can potentially cause problems and therefore require correction.
TIPS	Tips issues can improve the code usability or efficiency when corrected.

SUMMARY

The codes used in this Audit can be found on GitHub (https://github.com/nerdy-star/nerdy_smart_contract/tree/fd914ea5731494c1c0d291352495e4d26f196fa4).

The last commit of the code used for this Audit is "fd914ea5731494c1c0d291352495e4d26f196fa4".

Issues

HAECHI AUDIT found 0 critical issues, 1 major, and 1 minor issue. There are 4 Tips issues explained that would improve the code's usability or efficiency upon modification. We have verified that all issues have been fixed.

Severity	Issue	Status
⚠ MAJOR	Block dependent information is used as seed to generate random value.	(Resolved - v1.1)
MINOR	Unintended count update may occur on the *CharacterInfo* if *isValid* value is set to false.	(Resolved - v1.1)
TIPS	Unnecessary require code on the *ERC1155LUXON.burn* function.	(Resolved - v1.1)
TIPS	Unnecessary memory variable allocation on the *ERC721LUXON.mintLuxOn* function.	(Resolved - v1.1)
TIPS	Several important variable changes have no event emissions.	(Resolved - v1.1)
TIPS	Out of Gas error may be expected on *airdropMany* function by using a number of *AirdropInfo* parameters	(Resolved - v1.1)

OVERVIEW

Scope

```
├── Admin
│   ├── AirdropGachaTicket.sol
│   ├── LuxOnAdmin.sol
│   ├── LuxOnService.sol
│   └── data
│       ├── AirdropUser.sol
│       ├── CharacterData.sol
│       └── GachaData.sol
└── LUXON
    ├── myPage
    │   ├── centralization
    │   │   └── ERC721Centralization.sol
    │   ├── character
    │   │   └── LCT.sol
    │   └── inventory
    │       └── GachaTicket.sol
    ├── store
    │   └── desperado
    │       └── GachaMachineByGachaTicket.sol
    └── utils
        ├── ERC1155LUXON.sol
        ├── ERC721LUXON.sol
        ├── FindTokenList.sol
        ├── IERC721LUXON.sol
        ├── IGacha.sol
        ├── IGachaTicket.sol
        ├── LuxOnLive.sol
        └── LuxOnSuperOperators.sol
```

Access Controls

Airdrop/Gacha contracts have the following access control mechanisms.

- onlyOwner()
- onlySuperOperator()

onlyOwner() : modifier that controls access to variables including contract addresses that communicate with each other, operator address, and settings related to Airdrop and gacha.

- AirdropGachaTicket#setMintAddress()
- AirdropGachaTicket#setGachaDataAddress()
- AirdropGachaTicket#setAirdropUserAddress()
- AirdropGachaTicket#setAirdropRemainCount()
- AirdropGachaTicket#addAirdropRemainCount()
- AirdropGachaTicket#subAirdropRemainCount()
- AirdropGachaTicket#airdrop()
- AirdropGachaTicket#airdropMany()
- LuxOnAdmin#setSuperOperator()
- LuxOnService#setInspection()
- CharacterData#setCharacterData()
- CharacterData#deleteCharacterData()
- GachaData#setGachaInfo()
- GachaData#setGachaInfos()
- GachaData#removeGachaInfo()
- ERC721Centralization#addToeknAddresses()
- GachaMachineByGachaTicket#setMintAddress()
- GachaMachineByGachaTicket#setGachaDataAddress()
- GachaMachineByGachaTicket#withdraw()
- ERC721LUXON#setBaseURI()
- ERC1155LUXON#setName()
- ERC1155LUXON#setSymbol()
- ERC1155LUXON#setURI()
- LuxOnLive#setLuxOnService()
- LuxOnSuperOperators#setLuxOnAdmin()
- LuxOnSuperOperators#setOperator()

onlySuperOperator() : modifier that controls access to mint and burn functions and ownership change functions.

- AirdropUser#setAirdropUserInfo()
- AirdropUser#addAirdropUserInfo()
- AirdropUser#subAirdropUserInfo()
- AirdropUser#airdrop()
- ERC721Centralization#setRealOwnerOnce()
- ERC721Centralization#setRealOwner()
- ERC721Centralization#transferCenter()
- ERC721Centralization#setCentralizationDataOnceWithLog()
- ERC721Centralization#setCentralizationDataOnce()
- ERC721Centralization#setCentralizationDataWithLog()
- ERC721Centralization#setCentralizationData()
- ERC721Centralization#decentralizationDataOnce()
- ERC721Centralization#decentralizationDataOnceWithLog()
- ERC721Centralization#decentralizationData()
- ERC721Centralization#decentralizationDataWithLog()
- LCT#mintLuxOn()
- LCT#burn()
- ERC1155LUXON#burn()
- ERC1155LUXON#burnBatch
- ERC1155LUXON#mint
- ERC1155LUXON#mintBatch

The owner and operator have permissions that can change the crucial part of the system. It is highly recommended to maintain the private key as securely as possible and strictly monitor the system state changes.

Project Overview

- Airdrop

The contracts in scope of audit are implementation of Airdrop and Gacha systems. The project team set the user's address and token ID, amount before the airdrop(AirdropUser.sol). Based on the information set by the project team, the Gacha Ticket is distributed to the user(AirdropGachaTicket.sol). The Gacha Ticket is an ERC1155 token implementation(GachaTicket.sol) and the user can use this ticket to make use of the Gacha system.

- Gacha

By burning the Gacha ticket, the user can mint a character that has a random ID value. The tier information(GachaData.sol) set by the project team makes the character's tier in proportion to tier information(GachaMachineByGachaTicket.sol). The character NFT(ERC721) based on the character ID(LCT.sol) is minted to the user.

- Pausable

The project team has the privilege to pause the Gacha process. If the state of the contract is in Inspection, users can not use the Gacha feature(LuxOnService.sol).

FINDINGS

⚠ MAJOR

Block dependent information is used as seed to generate random value.

```
    function randomTier(
        uint256 seed,
        uint256[] memory tierRatio,
        uint256 tierRatioSum
    ) public view returns (uint256 index) {
        uint256 ratio = uint256(
            keccak256(abi.encodePacked(block.timestamp, msg.sender, seed))
        ) % tierRatioSum;
        index = 0;
        uint256 ratioSum = 0;
        for (uint256 i = 0; i < tierRatio.length; i++) {
            ratioSum += tierRatio[i];
            if (ratio <= ratioSum) {
                break;
            }
            index++;
        }
    }

    function randomCharacterNum(uint256 _tier, uint256 seed)
        public
        view
        returns (string memory)
    {
        uint256 characterCount = DspCharacterData(characterDataAddress)
            .getCharacterCountByTire(_tier + 1);
        uint256 index = uint256(
            keccak256(abi.encodePacked(block.timestamp, msg.sender, seed * 2))
        ) % characterCount;
        return
            DspCharacterData(characterDataAddress)
                .getCharacterInfoByTireAndIndex(_tier + 1, index);
    }
```

[https://github.com/nerdy-star/nerdy_smart_contract/blob/haechi-labs/airdrop-gacha-ticket/contracts/LUXON/store/desperado/GachaMachineByGachaTicket.sol]

Issue

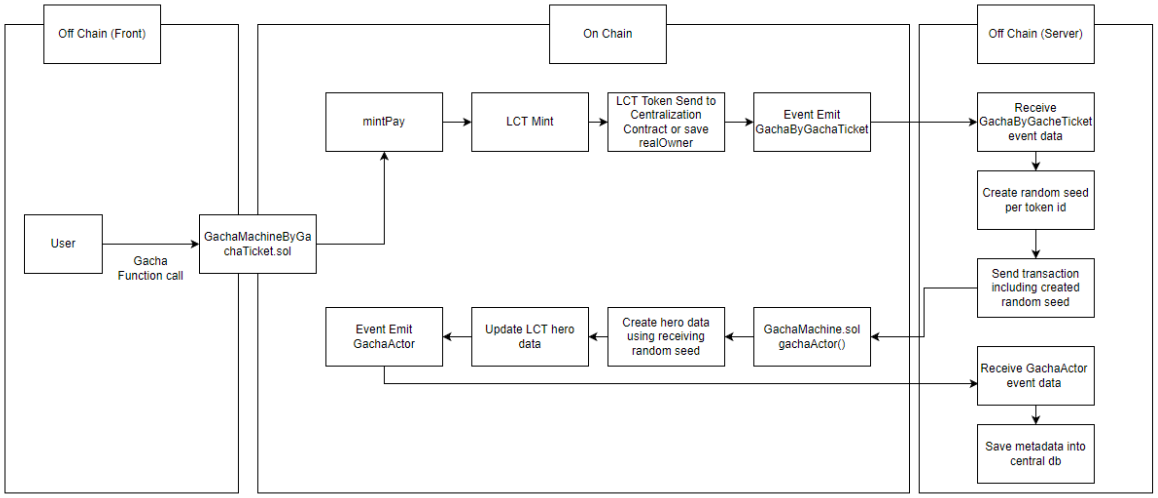
randomTier() and *randomCharacterNum()* functions of *GachaMachineByGachaTicket* contract generate a random number in the process of character generation. To randomize the tier information and index id, the functions make use of *block.timestamp* as a seed value. It is possible for a malicious user to simulate the Gacha process and predict the character ID to get a high-tier character NFT. Furthermore, the block information(*block.timestamp*, *block.coinbase*, *block.difficulty* etc.) is modifiable by miners and should not be used as the seed value of the random generation process.

Recommendation

The random number generation process needs to be reconsidered. Please refer to Appendix A.

Update

The off-chain random seed generation method is implemented to get rid of the predictability of random character IDs. The server listening event emission of gacha ticket burning generates and delivers a random seed to the gacha machine then updates new character information on the chain. [f29ee6aa]



MINOR

Unintended count update may occur on the *CharacterInfo* if isValid value is set to false.

```
function setCharacterData(CharacterInfo[] memory _characterData) external onlyOwner {
        for (uint256 i = 0; i < _characterData.length; i++) {
            CharacterInfo memory characterData_ = _characterData[i];
            if (!characterData[characterData_.name].isValid) {
                characterCount++;
            } else if (characterData[characterData_.name].tier != characterData_.tier) {
                uint256 index;
                uint256 _tier = characterData[characterData_.name].tier;
                for (uint256 j = 0; j < characterInfoTable[_tier].length; j++) {
                    if (keccak256(abi.encodePacked(characterInfoTable[_tier][j])) ==
keccak256(abi.encodePacked(characterData_.name))) {
                        index = i;
                        break;
                    }
                }
                for (uint256 j = index; j < characterInfoTable[_tier].length - 1; j++) {
                    characterInfoTable[_tier][j] = characterInfoTable[_tier][j + 1];
                }
                characterInfoTable[_tier].pop();
            }
            characterInfoTable[characterData_.tier].push(characterData_.name);
            characterData[characterData_.name] = characterData_;
        }
    }
```

[https://github.com/nerdy-star/nerdy_smart_contract/blob/haechi-labs/airdrop-gacha-ticket/contracts/Admin/data/CharacterData.sol]

Issue

The *characterCount* storage variable is increased by 1 if the *isValid* value of *characterInfo* is set false. However, there is no decrease in implementation if the *isValid* of *characterInfo* structure is true. It is possible to unintentionally increase the *characterCount* if specific *characterInfo*'s *isValid* set true and changed to false then again set to true.

Recommendation

Reverting the case when *isValid* value is false can prevent the *setCharacterData* function from changing the character count unintentionally.

Update

The validation logic of *isValid* value is implemented on the *setCharacterData* function.

```
function setCharacterData(CharacterInfo[] memory _characterData) external onlyOwner {
    for (uint256 i = 0; i < _characterData.length; i++) {
        require(_characterData[i].isValid, "isValid false use delete");
        if (!characterData[_characterData[i].name].isValid) {
            characterCount++;
        } else if (characterData[_characterData[i].name].tier != _characterData[i].tier) {
            uint256 index;
            uint256 _tier = characterData[_characterData[i].name].tier;
            uint256 _gachaGrade = characterData[_characterData[i].name].gachaGrade;
            for (uint256 j = 0; j < characterInfoTable[_tier][_gachaGrade].length; j++) {
                if (keccak256(abi.encodePacked(characterInfoTable[_tier][_gachaGrade][j])) ==
keccak256(abi.encodePacked(_characterData[i].name))) {
                    index = j;
                    break;
                }
            }
            for (uint256 j = index; j < characterInfoTable[_tier][_gachaGrade].length - 1; j++) {
                characterInfoTable[_tier][_gachaGrade][j] = characterInfoTable[_tier][_gachaGrade][j
+ 1];
            }
            characterInfoTable[_tier][_gachaGrade].pop();
        }

characterInfoTable[_characterData[i].tier][_characterData[i].gachaGrade].push(_characterData[i].name
);
        characterData[_characterData[i].name] = _characterData[i];

        emit SetCharacterData(_characterData[i].name, _characterData[i].tier,
_characterData[i].gachaGrade, _characterData[i].classType, _characterData[i].nation,
_characterData[i].element, _characterData[i].isValid);
    }
}
```

[https://github.com/nerdy-star/nerdy_smart_contract/blob/6b0a37003e7f91b153af22648de1bdf60f75c848/contracts/Admin/data/CharacterData.sol#L37]

💡 **TIPS**

Unnecessary require code on the ERC1155LUXON.burn function.

```
  function burn(address account, uint256 id, uint256 value) external virtual onlySuperOperator {
        require(
            account == _msgSender() || isSuperOperator(msg.sender) || isApprovedForAll(account,
_msgSender()),
            "ERC1155: caller is not owner nor approved"
        );

        _burn(account, id, value);
    }
```

[https://github.com/nerdy-star/nerdy_smart_contract/blob/haechi-labs/airdrop-gacha-ticket/contracts/LUXON/utils/ERC1155LUXON.sol]

Issue

Burning the *GachaTicket* is only allowed by *superOperator* because of the *onlySuperOperator* modifier. As general users cannot call the burn function directly, checking the ownership of *GachaTicket* is redundant and has no effect.

Recommendation

Removing the check code of the burn function can reduce the gas fee of the function call.

Update

The Redundant ownership check is removed from the *burn* function.

```
function burn(address account, uint256 id, uint256 value) external virtual onlySuperOperator {
    _burn(account, id, value);
}
```

[https://github.com/nerdy-star/nerdy_smart_contract/blob/6b0a37003e7f91b153af22648de1bdf60f75c848/contracts/LUXON/utils/ERC1155LUXON.sol#L53]

💡 **TIPS**

Unnecessary memory variable allocation on the *ERC721LUXON.mintLuxOn* function.

```
  function mintLuxOn(address mintUser, uint256 quantity, string[] memory characterId) external
onlySuperOperator {
        require(characterId.length == quantity, "quantity != gacha count");
        uint256 tokenId = _currentIndex;
        uint256[] memory tokenIds = new uint256[](quantity);
        for (uint8 i = 0; i < quantity; i++) {
            tokenIds[i] = tokenId;
            characterInfo[tokenId++] = characterId[i];
        }
        _safeMint(mintUser, quantity);
    }
```

[https://github.com/nerdy-star/nerdy_smart_contract/blob/haechi-labs/airdrop-gacha-ticket/contracts/LUXON/myPage/character/LCT.sol]

Issue

There is an unnecessary array variable(*tokenIds*) in memory allocation on the process of *mintLuxOn* function of *ERC721LUXON* Contract.

Recommendation

The tokenIds variable has no effect on the process and the result of the *mintLuxOn* function. Removing this variable can save the gas fee of the function call.

Update

The unnecessary variable is removed and the function name is changed to *mintByCharacterName*.

```
function mintByCharacterName(address mintUser, uint256 quantity, string[] memory characterName)
external onlySuperOperator {
    require(characterName.length == quantity, "quantity != gacha count");
    uint256 tokenId = nextTokenId();
    for (uint8 i = 0; i < quantity; i++) {
        emit MintByCharacterName(mintUser, tokenId, characterName[i]);
        characterInfo[tokenId++] = characterName[i];
    }
    _safeMint(mintUser, quantity);
}
```

[https://github.com/nerdy-star/nerdy_smart_contract/blob/6869b021a74dc073ab6bf79cefbd90eb87e78b37/contracts/LUXON/myPage/character/LCT.sol#L18]

💡 **TIPS**

Several important variable changes have no event emissions.

- `DspCharacterData#setCharacterData()`
- `DspCharacterData#deleteCharacterData()`
- `DspGachaData#setGachaInfo()`
- `DspGachaData#setGachaInfos()`
- `DspGachaData#removeGachaInfo()`
- `AirdropGachaTicket#setMintAddress()`
- `AirdropGachaTicket#setGachaDataAddress()`
- `AirdropGachaTicket#setAirdropUserAddress()`
- `AirdropGachaTicket#setAirdropRemainCount()`
- `AirdropGachaTicket#addAirdropRemainCount()`
- `AirdropGachaTicket#subAirdropRemainCount()`
- `AirdropGachaTicket#airdrop()`
- `AirdropGachaTicket#airdropMany()`
- `ERC721Centralization#addToeknAddresses()`
- `ERC721Centralization#setRealOwnerOnce()`
- `ERC721Centralization#setRealOwner()`
- `ERC721Centralization#transferCenter()`
- `GachaMachineByGachaTicket#setMintGoodsInfo()`
- `GachaMachineByGachaTicket#setMintAddress()`
- `GachaMachineByGachaTicket#setCentralizationAddress()`
- `GachaMachineByGachaTicket#setGachaDataAddress()`
- `GachaMachineByGachaTicket#setCharacterDataAddress()`
- `LuxOnLive#setLuxOnService()`
- `LuxOnSuperOperators#setLuxOnAdmin()`
- `LuxOnSuperOperators#setOperator()`

[Functions without event emission]

Issue

Several important variable changes have no event emissions.

Recommendation

Emitting events on the critical variable changes can enhance the visibility of the project.

Update

Event emissions are added on the commit [6b0a3700].

💡 **TIPS**

Out of Gas error may be expected on airdropMany function by using a number of AirdropInfo parameters.

```
      function airdropMany(AirdopInfo[] memory airdopInfo) external onlyOwner {
        for (uint256 i = 0; i < airdopInfo.length; i++) {
            require(airdropLimit[airdopInfo[i].tokenId] >= airdopInfo[i].amount, "total: The number
of air drops is insufficient.");
            airdropLimit[airdopInfo[i].tokenId] -= airdopInfo[i].amount;
            AirdropUser(airdropUserAddress).airdrop(mintAddress, airdopInfo[i].tokenId,
airdopInfo[i].to, airdopInfo[i].amount);
            DspGachaData.GachaInfo memory _gachaInfo =
DspGachaData(gachaDataAddress).getGachaInfo(airdopInfo[i].tokenId);
            require(_gachaInfo.isValid, "not valid token id");
            ERC1155LUXON(mintAddress).mint(airdopInfo[i].to, airdopInfo[i].tokenId,
airdopInfo[i].amount, '');
        }
    }
```

[https://github.com/nerdy-star/nerdy_smart_contract/blob/haechi-labs/airdrop-gacha-ticket/contracts/Admin/AirdropGachaTicket.sol]

Issue

The airdrop is conducted by the project team by minting every token by calling functions on-chain. This method has an advantage on the distribution of tokens to a small size of the user of intermittent minting but may cause an overrated gas fee and excess of the block gas limit.

The off-chain method using the Merkle tree may complement the current airdrop process. Furthermore, this could secure the transparency of the project by using a hash tree.

Recommendation

Refer to appendix B and consider the airdrop scale and use case to enhance the project's airdrop implementation.

Update

The merkle proof logic is added on the *AirdropUser* contract. [6b0a3700].

The Luxon team commented on this implementation as follows.

- To make airdropped users not to pay for their gas fee, validation logic of airdrop condition is called by the owner using *onlyOwner*
- A Centralized server pre-calculates the gas fee to prevent out-of-gas problems.

Appendix

A. Random number generation

Implementing random value generation on the smart contract is not a simple process because the seed value is observable to anyone. Specifically setting seed value using block-related attribution is risky because a malicious actor can predict the random value. (SWC-120)

The gachaCharacter function currently uses block.timestamp as seed value which can predict random gachaId in advance of the function call. There are different methods of implementing random value generation and we'd like to suggest the Chainlink VRF and the off-chain random value method.

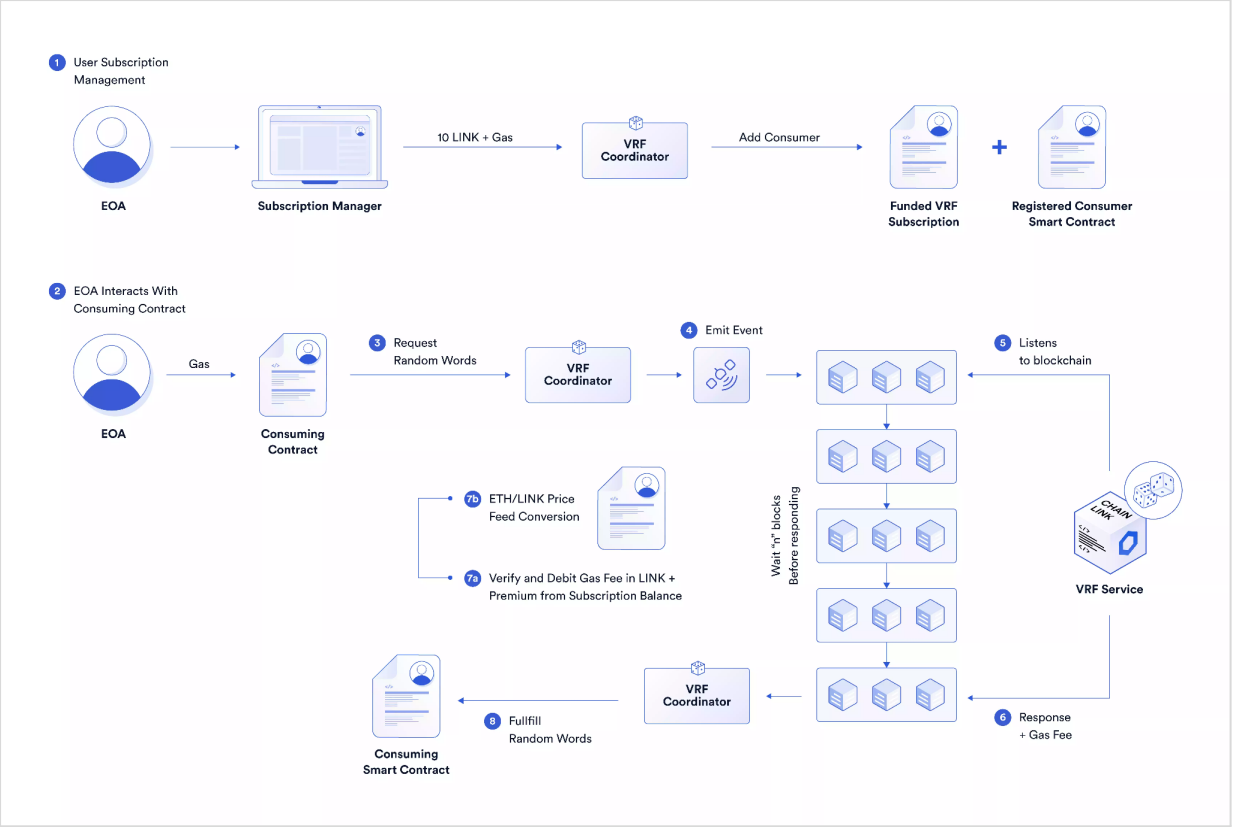
Chainlink VRF
Pros : Decentralized random number generation.
Cons : Fees are charged on random number generation.
(In polygon, 0.0005 Link : about 4.2 KRW)

Off-chain Custom Method
Pros : Relative simple structure of random number generation.
Cons : Minor transparency due to centralized random number generation.

Option 1 (Chainlink VRF)

Using Chainlink VRF allows you to achieve both decentralization and security of random value generation.

The operation of the Chainlink VRF is shown in the figure below after the project and the Chainlink VRF have been integrated.



[Chainlink VRF Overview from Chainlink docs]

Chainlink VRF Integration is well described in the following links:

[Exploring Chainlink VRF v2 | Developer Walkthrough]

[Get a Random Number | Chainlink Documentation]

[Programmatic Subscription | Chainlink Documentation]

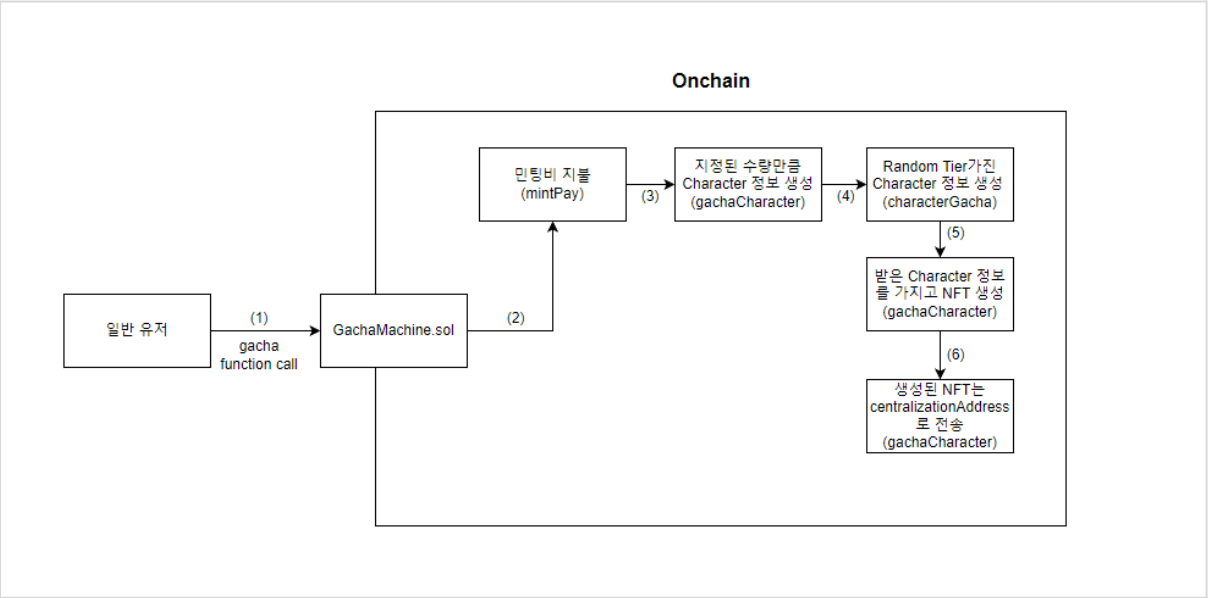
[VRF Security Considerations | Chainlink Documentation]

[VRF Best Practices | Chainlink Documentation]

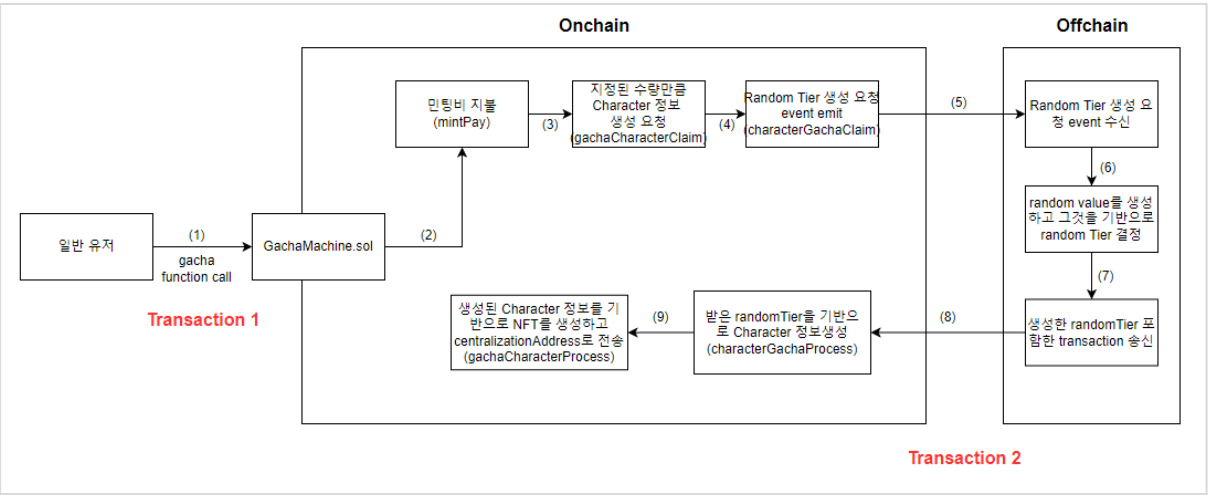
Option 2 (Custom Method)

An off-chain feed can be built and used to ensure the unpredictability of the Gacha system at the expense of decentralization.
Move the part of the transaction where the random number is created off-chain, and split the transaction in two so that the user cannot guess the random number.

Here's an illustration to help understanding. The first figure depicts the gacha execution flow inside the present project, the second depicts the Option 2 execution flow.



[Flow chart when a user generates a transaction that calls a gacha function in the current project]



[Option 2 Flow]

B. Airdrop using merkle tree

Construct a set of information that needs to be used for Gacha ticket minting(address, token id, amount, etc).

```
const testData = [
    ["0x1eff47bc3a10a45d4b230b5d10e37751fe6aa718", "1500001", "100"],
    ["0xe1ab8145f7e55dc933d51a18c793f901a3a0b276", "1500001", "10"],
    ["0xe57bfe9f44b819898f47bf37e5af72a0783e1141", "1500001", "15"],
    ["0xd41c057fd1c78805aac12b0a94a405c0461a6fbb", "1500001", "33"]
];
```

[Test Airdrop information]

Merkletreejs can be used to create a Merkle tree with each piece of information as a leaf node.

```
import { keccak256, solidityKeccak256 } from "ethers/lib/utils.js";
import { MerkleTree } from "merkletreejs";

const hashFn = (data) => keccak256(data).slice(2);

function createTree(targets) {
    const elements = targets.map((elem) =>
        solidityKeccak256(
            ["address", "uint256", "uint256"],
            [elem[0], elem[1], elem[2]]
        )
    );
    const tree = new MerkleTree(elements, hashFn, { sort: true });
    return tree;
};

function getAccountProof(
    tree,
    account,
    tokenId,
    amount
) {
    const element = solidityKeccak256(
        ["address", "uint256", "uint256"],
        [account, tokenId, amount]
    );
    return tree.getHexProof(element);
};

//Merkle Tree 생성
const merkletree = createTree(testData);

//Root hash를 확인 가능
merkletree.getHexRoot()

//특정 leaf의 merkle proof를 확인 가능
getAccountProof(merkletree, "0x1eff47bc3a10a45d4b230b5d10e37751fe6aa718", "1500001", "100")
```

[Merkle Tree 및 Merkle Proof 예시]

Set the root hash of the generated Merkle tree to Airdrop Contract.

```
contract MerkleTestContract is Test {
    bytes32 public merkleRoot =
        0x0000000000000000000000000000000000000000000000000000000000000000;

    mapping(address => uint256 => uint256) userClaimed;

    constructor(bytes32 root) {
        merkleRoot = root;
    }

    function setMerkleRoot(bytes32 nerMerkleRoot) external onlyOwner {
        merkleRoot = nerMerkleRoot;
    }
```

[Test Airdrop Contract 1]

The airdrop function is invoked by the user. To demonstrate that Merkle tree includes their information, users can utilize tokenId, amount, and merkleProof.

```
    function airdrop(uint256 tokenId, uint256 amount, bytes32[] memory merkleProof, uint256
claimAmount) public {
        address thisSender = msg.sender;
        require(
            verifyClaim(thisSender, tokenId, amount, merkleProof) == true,
            "invalid proof"
        );
        require(
            (claimed[thisSender][tokenId] + claimAmount) <= amount,
            "already claimed all airdrop tokens"
        );
        claimed[thisSender][tokenId] = claimed[thisSender][tokenId] + claimAmount;
        ERC1155LUXON(mintAddress).mint(thisSender, tokenId, claimAmount, '');
//
//            further logic like gacha data implementation
//            ...
    }
```

[Test Airdrop Contract 2]

To confirm that the user's airdrop information is correct, the processProof() method produces Roothash based on the information supplied and validates that the value matches the root hash contained in the Contract.

```
    function verifyClaim(
        address claimer,
        uint256 tokenId,
        uint256 amount,
        bytes32[] memory merkleProof
    ) private returns (bool) {
        bytes32 leaf = keccak256(abi.encodePacked(claimer, tokenId, amount));
        return verifyProof(merkleProof, merkleRoot, leaf);
```

```
    }

    function verifyProof(
        bytes32[] memory proof,
        bytes32 root,
        bytes32 leaf
    ) internal pure returns (bool) {
        return processProof(proof, leaf) == root;
    }

    function processProof(bytes32[] memory proof, bytes32 leaf)
        internal
        pure
        returns (bytes32)
    {
        bytes32 computedHash = leaf;
        for (uint256 i = 0; i < proof.length; i++) {
            bytes32 proofElement = proof[i];
            if (computedHash <= proofElement) {
                computedHash = keccak256(
                    abi.encodePacked(computedHash, proofElement)
                );
            } else {
                computedHash = keccak256(
                    abi.encodePacked(proofElement, computedHash)
                );
            }
        }
        return computedHash;
    }
```

[Test Airdrop Contract 3]

DISCLAIMER

This report does not guarantee investment advice, the suitability of the business models, and codes that are secure without bugs. This report shall only be used to discuss known technical issues. Other than the issues described in this report, undiscovered issues may exist such as defects on the main network. In order to write secure smart contracts, correction of discovered problems and sufficient testing thereof are required.

End of Document