verichains

*SECURITY AUDIT OF*

# RAGMON LEVEL MERGE


**Public Report**

*Jul 18, 2024*


## Verichains Lab

info@verichains.io

https://www.verichains.io

*Driving Technology > Forward*

# ABBREVIATIONS

| Name | Description |
|---|---|
| **Ethereum** | An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications. |
| **Ether (ETH)** | A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network. |
| **Smart contract** | A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract. |
| **Solidity** | A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform. |
| **Solc** | A compiler for Solidity. |
| **ERC20** | ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain. |

# EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jul 18, 2024. We would like to thank the Ragnarok: Monster World for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Ragmon Level Merge. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

**During the audit process, the audit team found some vulnerabilities in the given version of Ragmon Level Merge. Ragnarok: Monster World team has acknowledged and fixed all issues.**

# TABLE OF CONTENTS

**1. MANAGEMENT SUMMARY ..... 5**

**1.1. About Ragmon Level Merge ..... 5**

**1.2. Audit Scope ..... 5**

**1.3. Audit Methodology ..... 5**

**1.4. Disclaimer ..... 7**

**1.5. Acceptance Minute ..... 7**

**2. AUDIT RESULT ..... 8**

**2.1. Overview ..... 8**

**2.2. Findings ..... 9**

2.2.1. CRITICAL - Insufficient funds to request a random seed when payment is made in native token ..... 10

2.2.2. CRITICAL - The `setFeeReceiver` function can be called by anyone ..... 11

2.2.3. CRITICAL - Inconsistent between verifying fee and pay cost function ..... 12

2.2.4. HIGH - Allow users to call merge with same target monster before `VRFCoordinator` response ..... 13

2.2.5. HIGH - Incorrect handling of `msg.value` in contract ..... 15

2.2.6. MEDIUM - Unsafe transfer token ..... 18

2.2.7. LOW - The owner can set `rarityToBaseProbabilities[rarity]` out of range ..... 19

2.2.8. LOW - Lost user's monster when target monster is in material monster list ..... 20

2.2.9. LOW - Enable to lost funds when fee receiver is not set ..... 21

2.2.10. LOW - Missing gap storage in upgradeable contract ..... 21

**3. VERSION HISTORY ..... 22**

# 1. MANAGEMENT SUMMARY

## 1.1. About Ragmon Level Merge

In the immersive world of Ragnarok: Monster World, Ragmons stands as the digital embodiment of the monsters that players encounter.

The Ragmon Merge feature in `Ragnarok: Monster World` offers players the thrilling opportunity to upgrade their NFTs, marking a significant evolution in gameplay and strategy. This process involves merging multiple Ragmons of the same rarity, with a cost in Zeny (ZNY) or native token or other ERC20 token that varies depending on the rarity. When Ragmons are merged, there is a chance for the resulting Ragmon to be upgraded to the next level (e.g., from level 1 to level 2, level 2 to level 3, and so on up to the maximum level).

## 1.2. Audit Scope

This audit focused on identifying security flaws in code and the design of the smart contracts of Ragmon Level Merge.

It was conducted on latest commit `3753b87dc8ab41949f96744d05f795e89405805d` from git repository `https://github.com/zeroxand/ragmon-public-contracts/`.

The latest version of the following files were made available in the course of the review:

| SHA256 Sum | File |
|---|---|
| e1901b4adc5e2daef83351ff334e6e8709f165fc9a168efa957db54c8d7e35a8 | contracts/libraries/MergeUtil.sol |
| cd0c5401abaecb6b4ab42ffa88d28858f542cc3dd2f120c3cb75c82fb75203e3 | contracts/merge/LevelMerge.sol |
| ad4f669fbeca86a599bdb715794302de3496c78867f7982aa8cd513c625bc7bb | contracts/merge/MergeUpgradeable.sol |
| | |

## 1.3. Audit Methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

| SEVERITY LEVEL | DESCRIPTION |
|---|---|
| **CRITICAL** | A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately. |
| **HIGH** | A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority. |
| **MEDIUM** | A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed. |
| **LOW** | An issue that does not have a significant impact, can be considered as less important. |

*Table 1. Severity levels*

## 1.4. Disclaimer

Ragnarok: Monster World acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Ragnarok: Monster World understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Ragnarok: Monster World agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

## 1.5. Acceptance Minute

This final report served by Verichains to the Ragnarok: Monster World will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Ragnarok: Monster World, the final report will be considered fully accepted by the Ragnarok: Monster World without the signature.

# 2. AUDIT RESULT

## 2.1. Overview

The Ragmon Level Merge was written in `Solidity` language, with the required version to be `0.8.20`.

The smart contract facilitates the merging of in-game `Ragmon` by upgrading them using materials and specific probabilities. It integrates with the `IRagmon` and `INyangKit` interfaces, leveraging `MergeUtil` for calculations and `MergeUpgradeable` for core functionality. The contract maintains mappings for tracking `VRF` (Verifiable Random Function) requests, base probabilities for different rarities, and treasury rates. It includes methods for setting these base probabilities and treasury rates, executing merges, and calculating success rates and accumulated values. The merging process involves validating ownership, calculating costs and success rates, burning tokens, and handling `VRF` callbacks to determine merge success or failure.

## 2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of Ragmon Level Merge. Ragnarok: Monster World team acknowledged and fixed these issues according to Verichains's draft report. The following table shows the vulnerabilities found during the audit:

| # | Issue | Severity | Status |
|---|---|---|---|
| **1** | Insufficient funds to request a random seed when payment is made in native token | CRITICAL | FIXED |
| **2** | The `setFeeReceiver` function can be called by anyone | CRITICAL | FIXED |
| **3** | Inconsistent between verifying fee and pay cost function | CRITICAL | FIXED |
| **4** | Allow users to call merge with same target monster before `VRFCoordinator` response | HIGH | FIXED |
| **5** | Incorrect handling of `msg.value` in contract | HIGH | FIXED |
| **6** | Unsafe transfer token | MEDIUM | FIXED |
| **7** | Lost user's monster when target monster is in material monster list | LOW | FIXED |
| **8** | The owner can set `rarityToBaseProbabilities[rarity]` out of range | LOW | FIXED |
| **9** | Enable to lost funds when fee receiver is not set | LOW | FIXED |
| **10** | Missing gap storage in upgradeable contract | LOW | FIXED |

*Table 2. Vulnerability List*

### 2.2.1. CRITICAL - Insufficient funds to request a random seed when payment is made in native token

#### Position

- contracts/merge/LevelMerge.sol#L113

#### Description

The `requestRandomSeed` function requires a specific quantity of native token. If the payment method is native token, the contract checks the `msg.value` to be equal merge cost. The cost is then transferred to the fee receiver.As a result, the contract will have insufficient funds to request a random seed.

```
function _upgradeMonster(
    uint256 targetTokenId,
    uint256[] memory tokenIds
) internal {
   //...

    uint16 successRate = MergeUtil.calculateSuccessRate(
      baseProbabilities,
      RAGMON.monsters(targetTokenId).level,
      materialLevels
    );

    if (successRate == 10_000) {
      //...
    } else {
      bytes32 requestHash = _requestRandomness(
        msg.value, // @auditor: msg.value was used as a cost so the contract will not have enough funds to request a random seed
        callbackGasLimit,
        gasPrice,
        msg.sender
      );
    //...
}
```

The `msg.value` is used as a merging cost in here:

```
function payCost(uint256 amount) internal {
    require(amount > 0, 'Merge cost must be greater than 0');

    if (PAYMENT_TOKEN == address(0)) {
      require(msg.value == amount, 'Invalid ETH amount');
      payable(FEE_RECEIVER).transfer(amount);
    } else if (
      IERC165(PAYMENT_TOKEN).supportsInterface(type(IZeny).interfaceId)
    ) {
      //...
```

```
    } else {
      //...
    }
  }
```

### RECOMMENDATION

When the payment method is native token, the contract should ensure `msg.value` is equal to the total cost of merging and requesting a random seed.

### UPDATES

- ***Jul 16, 2024***: The issue has been acknowledged and fixed by the Ragnarok: Monster World team.

### 2.2.2. CRITICAL - The `setFeeReceiver` function can be called by anyone

**Position**

- `contracts/merge/MergeUpgradeable.sol`#L66

**Description**

The `setFeeReceiver` function allows anyone to change the fee receiver address. Attacker can change the fee receiver address to their own address and receive the cost of merging.

```
function setFeeReceiver(address _feeReceiver) external {
    require(_feeReceiver != address(0), 'Invalid fee receiver');

    FEE_RECEIVER = _feeReceiver;
}
```

### RECOMMENDATION

The `setFeeReceiver` function should be restricted to the owner only.

### UPDATES

- ***Jul 16, 2024***: The issue has been acknowledged and fixed by the Ragnarok: Monster World team.

### 2.2.3. CRITICAL - Inconsistent between verifying fee and pay cost function

#### Position

- contracts/merge/LevelMerge.sol#L91-L96

#### Description

The contract verifies that `msg.value` is greater than or equal to the estimated fee and merge cost in native tokens. However, the `payCost` function collects the payment using `PAYMENT_TOKEN`. If `PAYMENT_TOKEN` is not the native token, the user is charged both the native token and the specified token, leading to a loss of funds for the user.

```
function merge(
    uint256 targetTokenId,
    uint256[] memory materialTokenIds,
    uint16 berryAmount
    ) external payable nonReentrant whenNotPaused {
        //...
        uint256 mergeCost = mergeCostByRarity[rarity];
        uint256 estimatedFee = IRoninVRFCoordinatorForConsumers(vrfCoordinator)
          .estimateRequestRandomFee(callbackGasLimit, gasPrice);
        if (msg.value < estimatedFee + mergeCost) {
          revert InsufficientFee();
        }
        //...
        _upgradeMonster(...);
    }

    function _upgradeMonster(...){
        payCost(mergeCost);

        //...
    }

    function payCost(uint256 amount) internal {
        require(amount > 0, 'Merge cost must be greater than 0');

        if (PAYMENT_TOKEN == address(0)) {
            require(msg.value >= amount, 'Invalid ETH amount');
            payable(FEE_RECEIVER).transfer(amount);
        } else if (
            IERC165(PAYMENT_TOKEN).supportsInterface(type(IZeny).interfaceId)
        ) {
            IZeny(PAYMENT_TOKEN).burnFrom(msg.sender, amount);
        } else {
            IERC20(PAYMENT_TOKEN).transferFrom(msg.sender, FEE_RECEIVER, amount);
        }
    }
```

### RECOMMENDATION

The contract should move the `msg.value` verification condition into the `payCost` function to ensure consistent payment handling.

### UPDATES

- ***Jul 16, 2024***: The issue has been acknowledged and fixed by the Ragnarok: Monster World team.

#### 2.2.4. HIGH - Allow users to call merge with same target monster before `VRFCoordinator` response

**Position**

- `contracts/merge/LevelMerge.sol`#merge()

**Description**

During the merge process, users use material monsters to try to upgrade a target monster. The success rate of this upgrade is based on the target monster's current level, with higher levels having lower success rates.

While waiting for a fulfillment from the `VRFCoordinator`, if the success rate is less than `10,000`, the target monster is not locked, and its status, including its level, remains unchanged.

If the `merge` function is called again with the same target monster before the `VRFCoordinator` fulfills, the success rate is calculated using the monster's previous level instead of its new level (if it had been upgraded). This allows users to get higher success rates than intended. As a result, this weakens the game's balance, potentially causing economic and gameplay issues.

*Attacker Scenario:*

- The attacker has a low-level target monster (e.g., level 1) and several material monsters.
- The attacker starts a merge operation, sending a request for randomness to the `VRFCoordinator`.
- Before the `VRFCoordinator` responds, the attacker quickly calls the `merge` function again with the same target monster and possibly different material monsters.
- If the first response from the `VRFCoordinator` is successful, the target monster is upgraded. However, the second response should use the success rates based on the target monster's new level. Unfortunately, the success rates are determined using the target monster's previous level, which is higher than intended.

```
function merge(
    uint256 targetTokenId,
    uint256[] memory materialTokenIds,
```

```
    uint16 berryAmount
    ) external payable nonReentrant whenNotPaused {
        //...

        uint8 targetLevel = RAGMON.monsters(targetTokenId).level;
        uint16 baseSuccessRate = MergeUtil.calculateSuccessRate(
            baseProbabilities,
            targetLevel,
            materialLevels
        );
        //...
        uint16 additionalSuccessRate = baseProbabilities[targetLevel] * berryAmount;
        uint16 successRate = baseSuccessRate + additionalSuccessRate;
        //...
        _upgradeMonster(...);
    }
    function _upgradeMonster(...){
        //...
        if (successRate >= 10_000) {
        //...
        } else {
            bytes32 requestHash = _requestRandomness(
            msg.value - mergeCost,
            callbackGasLimit,
            gasPrice,
            msg.sender
            );

            requestHashToReceiver[requestHash] = msg.sender;
            requestHashToTargetTokenId[requestHash] = targetTokenId;
            //...
            requestHashToSuccessRate[requestHash] = successRate;
            //...
        }
    }
    function _fulfillRandomSeed(
        bytes32 requestHash,
        uint256 randomSeed
    ) internal override {
        uint256 tokenId = requestHashToTargetTokenId[requestHash];
        address receiver = requestHashToReceiver[requestHash];
        //...
        uint16 successRate = requestHashToSuccessRate[requestHash];

        bool success = (randomSeed % 10_000) < successRate;
        if (success) {
            RAGMON.upgrade(tokenId);
        } else {
            //...
        }
```

```
        emit MonsterMerged({
            owner: receiver,
            targetTokenId: tokenId,
            materialTokenIds: materialTokenIds,
            successRate: successRate,
            upgraded: success
        });
    }
```

**RECOMMENDATION**

The contract should lock all the monsters and related states in storage while waiting for fulfillment from the `VRFCoordinator`.

**UPDATES**

- ***Jul 17, 2024***: The issue has been acknowledged and fixed by the Ragnarok: Monster World team.

### 2.2.5. HIGH - Incorrect handling of `msg.value` in contract

**Position**:

- `contracts/merge/MergeUpgradeable.sol`#L181
- `contracts/merge/MergeUpgradeable.sol`#L192

**Description**:

When upgrading a monster successfully without using `VRF`, the contract subtracts the `mergeCost` from `msg.value` and transfers it to the user without verifying the payment token. If the payment token is not the native token, the contract will revert. Additionally, if the user mistakenly sends more than the required `mergeCost`, the contract does not refund the excess amount.

If the merge process requires using the `VRF`, the contract uses any excess `msg.value` to pay for the `VRF` fee. Again, if the payment token is not the native token, the contract will revert.

```
function merge(
    uint256 targetTokenId,
    uint256[] memory materialTokenIds,
    uint16 berryAmount
  ) external payable nonReentrant whenNotPaused {
    //...

    uint256 mergeCost = mergeCostByRarity[rarity];
    uint256 estimatedFee = IRoninVRFCoordinatorForConsumers(vrfCoordinator)
      .estimateRequestRandomFee(callbackGasLimit, gasPrice);
```

```
    uint256 totalFee = PAYMENT_TOKEN == address(0)
      ? estimatedFee + mergeCost
      : estimatedFee;
    if (msg.value < totalFee) { // @auditor: The contract should check msg.value ==
totalFee
      revert InsufficientFee();
    }
    //...

    _upgradeMonster(
      monster,
      targetTokenId,
      materialTokenIds,
      mergeCost,
      totalValue,
      successRate
    );
  }

    function _upgradeMonster(
        IRagmon.Monster memory monster,
        uint256 targetTokenId,
        uint256[] memory materialTokenIds,
        uint256 mergeCost,
        uint256 totalValue,
        uint16 successRate
    ) internal {
        payCost(mergeCost);
        //...

        if (successRate >= 10_000) {
            //...

            payable(msg.sender).transfer(msg.value - mergeCost); // @auditor: Must check
payment token before subtracting mergeCost

            //...
        } else {
            bytes32 requestHash = _requestRandomness(  // @auditor: Must pass estimatedFee
instead of msg.value - mergeCost
                msg.value - mergeCost,
                callbackGasLimit,
                gasPrice,
                msg.sender
            );
            //...
        }
    }

    function payCost(uint256 amount) internal {
```

```
        require(amount > 0, 'Merge cost must be greater than 0');

        if (PAYMENT_TOKEN == address(0)) {
            require(msg.value >= amount, 'Invalid ETH amount');
            payable(FEE_RECEIVER).transfer(amount);
        } else if (
            IERC165(PAYMENT_TOKEN).supportsInterface(type(IZeny).interfaceId)
        ) {
            IZeny(PAYMENT_TOKEN).burnFrom(msg.sender, amount);
        } else {
            IERC20(PAYMENT_TOKEN).safeTransferFrom(msg.sender, FEE_RECEIVER, amount);
        }
    }
```

## RECOMMENDATION

When calling `VRF`, the contract should pass the exact estimated fee (e.g., `estimatedFee`) instead of `msg.value - mergeCost`. Moreover, the contract must check payment token in refund logic.

## UPDATES

- ***Jul 18, 2024***: The issue has been acknowledged and fixed by the Ragnarok: Monster World team.

### 2.2.6. MEDIUM - Unsafe transfer token

#### Position

- contracts/merge/MergeUpgradeable.sol#L90

#### Description

When the payment token is not native token and Zeny token, the contract use `transferFrom` method to collect token from the user. If the token does not revert when the transfer fails (e.g., due to insufficient balance or insufficient allowance) and only return a true/false value, the contract will not handle the failure and continue execution. This can lead to unexpected behavior and loss of tokens.

```
function payCost(uint256 amount) internal {
        //...

        if (PAYMENT_TOKEN == address(0)) {
            //...
        } else if (
            IERC165(PAYMENT_TOKEN).supportsInterface(type(IZeny).interfaceId)
        ) {
            //...
        } else {
            IERC20(PAYMENT_TOKEN).transferFrom(msg.sender, FEE_RECEIVER, amount);
        }
    }
```

#### RECOMMENDATION

Use the `SafeERC20` library to handle token transfers. The `SafeERC20` library provides a set of functions that prevent the contract from continuing execution if the token transfer fails.

#### UPDATES

- ***Jul 16, 2024***: The issue has been acknowledged and fixed by the Ragnarok: Monster World team.

### 2.2.7. LOW - The owner can set `rarityToBaseProbabilities[rarity]` out of range

#### Position

- `contracts/merge/LevelMerge.sol`#setBaseProbabilities()

#### Description

The `setBaseProbabilities` function allows the owner to set the base probabilities of each rarity. As the comment, the base probabilities is in the range of 1 to 10000. However, the contract does not check the base probabilities before setting them. This can lead to unexpected revert or behavior.

```
function setBaseProbabilities(
        Rarity rarity,
        uint16[] memory baseProbabilities
    ) external onlyOwner {
        require(
            RAGMON.maxLevel(rarity) == baseProbabilities.length,
            'Invalid probabilities'
        );

        rarityToBaseProbabilities[rarity] = baseProbabilities; // @auditor:
baseProbabilities can be out of range
    }
```

#### RECOMMENDATION

The contract should check the base probabilities of each rarity before setting them.

#### UPDATES

- ***Jul 16, 2024***: The issue has been acknowledged and fixed by the Ragnarok: Monster World team.

### 2.2.8. LOW - Lost user's monster when target monster is in material monster list

#### Position

- contracts/merge/LevelMerge.sol#merge()
- contracts/merge/LevelMerge.sol#_upgradeMonster()

#### Description

When a user calls the `merge` function with a target monster that is also included in the material monster list, the target monster is lost. This occurs because the material monster list is not checked for the target monster inclusion.

```
function merge(
    uint256 targetTokenId,
    uint256[] memory materialTokenIds,
    uint16 berryAmount
    ) external payable nonReentrant whenNotPaused {
        //...
        _upgradeMonster(...);
    }
    function _upgradeMonster(...){
        //...
        for (uint256 i = 0; i < materialTokenIds.length; i++) {
            uint256 tokenId = materialTokenIds[i];
            require(RAGMON.ownerOf(tokenId) == msg.sender, 'Not owner');
            require(
            RAGMON.monsters(tokenId).monsterType == monster.monsterType,
            'Not same monster type'
            );

            RAGMON.burn(tokenId);
        }
        //...
    }
```

#### RECOMMENDATION

The contract should check the material monster list for the target monster's inclusion and revert the transaction if the target monster is found in the material monster list.

#### UPDATES

- ***Jul 17, 2024***: The issue has been acknowledged and fixed by the Ragnarok: Monster World team.

### 2.2.9. LOW - Enable to lost funds when fee receiver is not set

**Position**

- `contracts/merge/MergeUpgradeable.sol`#payCost()

**Description**

Due to fee receiver does not set in initialization and the `payCost` function transfers the merging cost to the fee receiver, the merging cost will be lost if the fee receiver is not set.

**RECOMMENDATION**

The contract should check the fee receiver address before transferring the merging cost.

**UPDATES**

- ***Jul 16, 2024***: The issue has been acknowledged and fixed by the Ragnarok: Monster World team.

### 2.2.10. LOW - Missing gap storage in upgradeable contract

**Position**

- `contracts/merge/MergeUpgradeable.sol`
- `contracts/vrf/VRFConsumerUpgradeable.sol`

**Description**

When designing an upgradeable smart contract using proxies, the storage layout must remain consistent across all contract versions. If additional storage variables are added in future contract versions without reserved space, it can lead to storage collisions with existing variables, potentially leading to critical issues such as data corruption or loss.

**RECOMMENDATION**

Recommend adding appropriate storage gap at the end of upgradeable contracts such as the below. Please reference OpenZeppelin upgradeable contract templates.

```
uint256[50] private __gap;
```

**UPDATES**

- ***Jul 16, 2024***: The issue has been acknowledged and fixed by the Ragnarok: Monster World team.

# 3. VERSION HISTORY

| Version | Date | Status/Change | Created by |
|---|---|---|---|
| **1.0** | *Jul 16, 2024* | Public Report | Verichains Lab |
| **1.1** | *Jul 17, 2024* | Public Report | Verichains Lab |
| **1.2** | *Jul 18, 2024* | Public Report | Verichains Lab |

*Table 3. Report versions history*