SECURITY AUDIT OF

METASPETS SMART CONTRACTS

Public Report

Apr 21, 2022

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 prepared by Verichains Lab on Apr 21, 2022. We would like to thank the MetaSpets 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 MetaSpets Smart Contracts. 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 had identified some vulnerable issues in the smart contract code, along with some recommendations. MetaSpets team has resolved and updated most of the issues following our recommendations.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About MetaSpets Smart Contracts

MetaSpets is a Turn-based Idle RPG in a magical post-apocalyptic setting. You will meet wondrous sentient pets who can become your companions and guardians, and together venture into the fantastical new world to claim extraordinary rewards.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of MetaSpets Smart Contracts. It was conducted on commit 5056d2ac1bd7b3a37cbbd78a809fdf595a5c1b9c from git repository *https://gitlab.com/MetaSpets/contracts/*.

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

SHA256 Sum	File
cf87b9d55199bdf700bfbc15bfc4299b1f25f3a24ee21f4a742bf711ca49ac59	**MSPIdo.sol**
c52ab34ed7295f70fa3d3dfe88697d858d7bfcda151dc544abf934d07ce12fc8	**MSPVestingFactory.sol**
843339809b7f5736613260a5f1974c230beb9380a8af98a06b326c46aee6a58b	**MSP.sol**
552665ed2a114cee027da2636781c724070b435e312d5c3cd336909c219c2542	**MSPPrivateSaleVesting.sol**
c4155df1b3344691dfee64de3d81c95672dfce67d1e55fa647e174325332ab65	**MSPPublicSaleVesting.sol**
6615a4018719d54ce123bb44a9e60ab56e3ce5c6c402660d3a4ab4817e088dd9	**MSPVesting.sol**
ef07f641beac305d688e3405756557d4fb294e343df7d22a5afdf631c63fa904	**EggBasketDetails.sol**
b1a57d459e513e25eb1dd24b31d17b6e68c37f731cb0b202032057cf9bc2a199	**EggBasket.sol**
4e6d6810ebd6d11cfbcb5d59a41eef0a611a50f3c665bfad2711df97586f77d1	**MSPAirdrop.sol**
b199b1d9719bcc706a1829dc7170b4b50d9271c003fcc79b25a3442ec2e660b7	**IEggBasket.sol**
82aebf3b497a898212a42672b2d2662bebff25a88a9571d7e39a390fd82b7c98	**IPetToken.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 was 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

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract.

However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The MetaSpets Smart Contracts was written in Solidity language, with the required version to be ^0.8.0. The source code was written based on OpenZeppelin's library.

There are three main parts in the audit scope as shown in the below section:

2.1.1. ERC20 token

MSP token is the main ERC20 token in Meta Super Pets (MetaSpets) game. The MSP contract extends ERC20. When the token contract is initialized, all the tokens (max supply amount) will be minted and transferred to the contract deployer wallet.

The below table lists some properties of the audited MSP contract (as of the report writing time).

PROPERTY	VALUE
Name	Meta Super Pet
Symbol	MSP
Decimals	18
Max Supply	500,000,000 (x10^{18})

Table 2. MSP token properties

2.1.2. Vesting and IDO contracts

The logic for the token vesting contract is defined in MSPVesting.sol, this contract implements a vesting mechanism to lock and release tokens according to a configured schedule defined by the contract owner. The token distribution process can be summarized as below:

- Before the start time, tokens are locked in the vesting contract when a new beneficiary is added.
- Once the start time is reached, a TGE amount will be unlocked and claimable.
- When the cliff time is reached, all the remaining tokens will be released at the end of each month.

The logic for the token IDO process is defined in MSPIdo.sol, the IDO contract can be in whitelist or first come first served mode. In whitelist mode, only whitelist users can buy the

IDO tokens. With first come first served mode, anyone can buy the IDO tokens. Users need to pay first, the tokens will be released later by the contract owner.

Note: The contract owner can withdraw tokens in the vesting and IDO contracts in case of an emergency situation.

2.1.3. Egg basket NFT and marketplace

The EggBasket contract in EggBasket.sol defines logic related to egg basket minting, buy/sell (marketplace), and egg basket opening. However, the randomness logic for eggs opening is defined in the PetToken contract, which is not in the scope of this audit.

2.2. Findings

During the audit process, the audit team had identified some vulnerabilities in the given version of MetaSpets Smart Contracts.

MetaSpets fixed the code, according to Verichains's draft report, in commit 2cd9ca394a3ac4072bc101105f819da05b050f7e.

2.2.1. MSPVesting.sol - Missing constraint checking for TGE and monthly release percentages LOW

In the constructor function of the MSPVesting contract, there is no restriction for _percentClaimAtTGE, _percentUnleasePerMonth, and _monthlyDuration variables.

```
constructor(
    address _token,
    address _owner,
    uint256 _vestingStartAt,
    uint256 _monthlyDuration,
    uint256 _percentClaimAtTGE,
    uint256 _vestingCliff,
    uint256 _percentUnleasePerMonth,
    uint256 _secondPerMonth
) {
    require(_token != address(0), "zero-address");
    require(_owner != address(0), "zero-address");
    MSPToken = IERC20(_token);
    _transferOwnership(_owner);
    vestingStartAt = _vestingStartAt;
    monthlyDuration = _monthlyDuration;
    percentClaimAtTGE = _percentClaimAtTGE;
    vestingCliff = _vestingCliff;
    percentUnleasePerMonth = _percentUnleasePerMonth;
    SECONDS_PER_MONTH = _secondPerMonth;
    monthlyStartAt = vestingStartAt.add(vestingCliff); // NOTE: the first…
   monthly claim with be 1 month (SECONDS_PER_MONTH) AFTER this timestamp…
  .
}
```

RECOMMENDATION

The following check should be added to the constructor function:

```
require(_percentClaimAtTGE + _monthlyDuration * _percentUnleasePerMonth <…
  = 100, "Invalid params")
```

UPDATES

- *Apr 21, 2022*: This issue has been acknowledged and fixed by MetaSpets team.

2.2.2. EggBasket.sol - WhiteList check should be skipped for contract owner LOW

This function is limited to contract owners only (users with DESIGNER_ROLE in this case), so the whiteList check is not necessary here. If whitelisted users want to mint tokens, they should call the whitelistMint function instead.

```
function ownerMint(uint256 count) external onlyRole(DESIGNER_ROLE) {
    require(count > 0, "No token to mint");
    address to = msg.sender;
    require(whiteList[to] >= count, "User not in whitelist or limit reach…
  ed"); // THIS CHECK SHOULD BE REMOVED
    require(tokenIdCounter.current() + count <= TOTAL_EGG, "Egg basket so…
  ld out");

    //address owner = address(this);
    // Transfer token
    //coinToken.transferFrom(to, owner, basketPrice * count);
    whiteList[to] -= count;

    for (uint256 i = 0; i < count; ++i) {
        uint256 id = tokenIdCounter.current();
        tokenIdCounter.increment();
        EggBasketDetails.BasketDetails memory basketDetail;
        basketDetail.id = id;
        basketDetail.index = i;
        basketDetail.price = 1000 * COIN_DECIMALS;
        basketDetail.egg_type = EGG_TYPE_BASKET;
        basketDetail.on_market = 0;
        basketDetail.owner_by = to;
        tokenDetails[id] = basketDetail;

        _safeMint(to, id);
        emit TokenCreated(to, id, id);
    }
    whiteListBought += count;
}
```

RECOMMENDATION

The above function can be updated as below:

```
function ownerMint(uint256 count) external onlyRole(DESIGNER_ROLE) {
    require(count > 0, "No token to mint");
    address to = msg.sender;
    require(tokenIdCounter.current() + count <= TOTAL_EGG, "Egg basket so…
  ld out");
    for (uint256 i = 0; i < count; ++i) {
        uint256 id = tokenIdCounter.current();
        tokenIdCounter.increment();
        EggBasketDetails.BasketDetails memory basketDetail;
        basketDetail.id = id;
        basketDetail.index = i;
        basketDetail.price = 1000 * COIN_DECIMALS;
        basketDetail.egg_type = EGG_TYPE_BASKET;
        basketDetail.on_market = 0;
        basketDetail.owner_by = to;
        tokenDetails[id] = basketDetail;

        _safeMint(to, id);
        emit TokenCreated(to, id, id);
    }
}
```

UPDATES

- *Apr 21, 2022*: This issue has been acknowledged and fixed by MetaSpets team.

2.2.3. EggBasket.sol - Basket price is mismatched with the default value LOW

In the mint function of the EggBasket contract, the value of the price field in the basketDetail struct is mismatched with the basketPrice variable.

```
function mint(uint256 count) external notContract {
    // ...

    for (uint256 i = 0; i < count; ++i) {
        uint256 id = tokenIdCounter.current();
        tokenIdCounter.increment();
        EggBasketDetails.BasketDetails memory basketDetail;
        basketDetail.id = id;
        basketDetail.index = i;
        // TODO check default value
        basketDetail.price = 1000 * COIN_DECIMALS; // MISMATCHED PRICE
        basketDetail.egg_type = EGG_TYPE_BASKET;
        basketDetail.on_market = 0;
        basketDetail.owner_by = to;
        tokenDetails[id] = basketDetail;
        _safeMint(to, id);
        emit TokenCreated(to, id, id);
    }

    boughtList[to] = boughtList[to] + count;
}
```

RECOMMENDATION

Complete the TODO as shown in the above comment.

UPDATES

- *Apr 21, 2022*: This issue has been acknowledged by MetaSpets team.

2.3. Additional notes and recommendations

2.3.1. MSPVesting.sol - Wrong description of monthlyDuration INFORMATIVE

The unit of monthlyDuration variable should be month, not second as shown in the below comment.

```
abstract contract MSPVesting is BEPOwnable {
    // ...
    // Vesting duration in seconds
    uint256 public monthlyDuration;
    // ...
```

UPDATES

- *Apr 21, 2022*: This issue has been acknowledged and fixed by MetaSpets team.

2.3.2. MSPVesting.sol - Redundant check in the calculateClaimable function INFORMATIVE

In the calculateClaimable function, the _now < monthlyStartAt check can be done with the elapsedMonths < 1 check.

```
function calculateClaimable(address _beneficiary) private view returns (u…
  int256, uint256, uint256) {
    uint256 _now = block.timestamp;

    // return 0 for any claim before the starting time
    if (_now < vestingStartAt) {
      return (0, 0, 0);
    }

    uint256 _tokenClaimable = 0;
    uint256 _tokenClaimedAtTGE = 0;

    Beneficiary storage bf = beneficiaries[_beneficiary];
    require(bf.initialBalance > 0, "beneficiary-not-found");

    _tokenClaimedAtTGE = bf.initialBalance.mul(percentClaimAtTGE).div(100…
  );

    // if the user has not ever claimed his _tokenClaimedAtTGE, add that …
  amount to be claimed
    if (bf.claimedAtTGE == 0) {
      _tokenClaimable =  _tokenClaimable.add(_tokenClaimedAtTGE);
    }

    // REDUNDANT CHECK HERE
    if (_now < monthlyStartAt) {
        return (0, _tokenClaimable, _tokenClaimedAtTGE);
    }
    uint256 elapsedTime = _now.sub(monthlyStartAt);
    uint256 elapsedMonths = elapsedTime.div(SECONDS_PER_MONTH);

    // If it does not pass the first month yet
    if (elapsedMonths < 1) {
      return (0, _tokenClaimable, _tokenClaimedAtTGE);
    }
    // ...
}
```

RECOMMENDATION

We can remove the redundant check below to make the calculateClaimable function more concise:

```
if (_now < monthlyStartAt) {
  return (0, _tokenClaimable, _tokenClaimedAtTGE);
}
```

2.3.3. EggBasket.sol, MSPIdo.sol - Unused OwnableUpgradeable INFORMATIVE

EggBasket and MSPIdo contracts inherit both the OwnableUpgradeable and AccessControlUpgradable contracts. However, the onlyOwner modifier of the OwnableUpgradable contract is not used anywhere in the code.

```
contract EggBasket is
    ERC721Upgradeable,
    AccessControlUpgradeable,
    UUPSUpgradeable,
    OwnableUpgradeable, // UNUSED
    IERC721Receiver,
    IEggBasket
{
    using EggBasketDetails for EggBasketDetails.BasketDetails;
    using Counters for Counters.Counter;
    // ...
}

contract MSPIdo is
AccessControlUpgradeable,
OwnableUpgradeable { // UNUSED

    using SafeMath for uint256;
    event BuyIdo(address indexed user, uint256 amount, uint256 buyAt);

    bytes32 public constant DESIGNER_ROLE = keccak256("DESIGNER_ROLE");
    bytes32 public constant WHITELIST_ROLE = keccak256("WHITELIST_ROLE");
    // ...
}
```

RECOMMENDATION

We can remove the OwnableUpgradeable contract if the AccessControlUpgradeable is already being used.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Apr 21, 2022*	Public Report	Verichains Lab

Table 3. Report versions history