verichains

SECURITY AUDIT OF

THE LAST WORLD SMART CONTRACTS

Public Report

March 09, 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 March 09, 2022. We would like to thank the The Last 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 The Last World 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 contracts code. The Last World team has resolved and updated all the recommendations.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About The Last World Smart Contracts

The last world is an MMORPG metaverses game about a fictional post-apocalypse world where humans come out of their bunker to recreate their civilization on the BSC chain with cool features and unique NFT characters for you to create and customize.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of The Last World Smart Contracts.

It was conducted on commit 6b7ec27787e759690b6f6f768b1e1f2c278394f2 from git repository *https://github.com/The-Last-World/token-contracts/*.

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

SHA256 Sum	File
465dd7857f06f10e4bea6639de3f92c208048f7ab6ae8088ba48b2bec2d5eb57	**BEP20UpgradeableProxy.sol**
c9c347cbe28db1e08970a3e58b9e7b26c6696539084a443153fb8fb5abc5f478	**IBEP20.sol**
c7445105782578840a7f84bd77c01734d889e56398ee849ed763904dcccb3869	**TLWBEP20Implementation.sol**
475dac337031e6fe242fee14ef0c079b9f9e323494f2a1c88380d9d529c7777c	**BEP20VestingVault.sol**
928c17296187d054cfcf252e45cb7fcc9a24e096019d5a5c77f36607035503f4	**interface/ApproveAndCallFallBack.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 The Last World Smart Contracts was written in Solidity language, with the required version to be ^0.6.0 (for the token contract) and ^0.8.12 (for the vesting contract). The source code was written based on OpenZeppelin's library.

There are two main parts in the audit scope. They are BEP20TokenImplementationV1 contract and TheLastWorldVestingVault contract.

2.1.1. BEP20TokenImplementationV1 contract

This is the ERC20 token contract for the The Last World game. Currently, the contract doesn't have the exact value of totalSupply. The contract implements mint public function with the onlyOwner modifier. So only the owner of the contract can mint new tokens when the _mintable property is set to true, which can change the totalSupply of the contract. Also, the contract implements the burnFrom function. So an allowance account can call this function to remove the owner balances.

In addition, this token contract is upgradable, so be aware that the admin can change its logic at any time.

Some other properties like token name, symbol, decimals,... cannot be listed here since they will be set dynamically at the deployment time.

2.1.2. TheLastWorldVestingVault contract

This is the vesting contract for the The Last World game, which extends the Ownable contract. With Ownable, by default, Contract Owner is also the contract deployer, but he can transfer ownership to another address at any time.

The token vesting schedule is added for each user by the contract owner. The token distribution process can be summarized as below:

- Before the cliff time, all tokens are locked in the vesting contract without TGE.
- Once the cliff time is reached, the tokens will be unlocked at the end of each month. The vesting duration is specified by the contract owner when adding the vesting schedule.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of The Last World Smart Contracts.

2.2.1. BEP20VestingVault.sol - Set wrong value for vesting startTime HIGH

In the addTokenGrant function, the startTime property of the token grant is set with a wrong value. The expression currentTime() + uint256(_vestingDurationInMonths) * 4 may indicate this is the end time of the vesting (not start time). The value for the startTime property should be passed from the function parameter.

```
function addTokenGrant(
    address _recipient,
    uint256 _amount,
    uint16 _vestingDurationInMonths,
    uint16 _vestingCliffInMonths
)
    external
    onlyOwner
{
    // ...
    Grant memory grant = Grant({
        startTime: currentTime() + uint256(_vestingDurationInMonths) * 4 …
  weeks, // ERROR
        amount: _amount,
        vestingDuration: _vestingDurationInMonths,
        monthsClaimed: 0,
        totalClaimed: 0,
        recipient: _recipient
    });
    tokenGrants[_recipient] = grant;
    emit GrantAdded(_recipient);
}
```

RECOMMENDATION

The code can be fixed as below:

```
function addTokenGrant(
    address _recipient,
    uint256 _amount,
    uint16 _vestingDurationInMonths,
    uint16 _vestingCliffInMonths,
    uint256 _startTime
)
    external
    onlyOwner
```

```
{
    // ...
    Grant memory grant = Grant({
        startTime: _startTime + uint256(_vestingCliffInMonths) * 4 weeks,…
    // FIXED
        amount: _amount,
        vestingDuration: _vestingDurationInMonths,
        monthsClaimed: 0,
        totalClaimed: 0,
        recipient: _recipient
    });
    tokenGrants[_recipient] = grant;
    emit GrantAdded(_recipient);
}
```

UPDATES

- *Mar 09,2022*: This issue has been acknowledged and fixed by the The Last World team in commit 2307fef823b6782e57ec6f59ad9dd04416b10b6c.

2.2.2. BEP20VestingVault.sol - Unclear logic for elapsedMonths MEDIUM

The current formula for elapsedMonths is added with 1 day, which means users can claim tokens 1 day early before the end of the month. The business logic is quite unclear here.

```
function calculateGrantClaim(address _recipient) private view returns (ui…
  nt16, uint256) {
    // ...
    // Check cliff was reached
    uint elapsedMonths = currentTime().sub(tokenGrant.startTime - 1 days)…
  .div(4 weeks);
    // ...
}
```

RECOMMENDATION

The dev team should review the logic for the elapsedMonths, make sure that users can claim 1 day early is necessary or not.

UPDATES

- *Mar 09,2022*: This issue has been acknowledged and fixed by the The Last World team in commit 2307fef823b6782e57ec6f59ad9dd04416b10b6c.

2.2.3. BEP20VestingVault.sol - Return wrong value for monthsVested MEDIUM

In the calculateGrantClaim function, when the vesting duration is ended, the returned value of the vested months is wrong. It should be subtracted by tokenGrant.monthsClaimed.

```
function calculateGrantClaim(address _recipient) private view returns (ui…
  nt16, uint256) {
    // ...
    // If over vesting duration, all tokens vested
    if (elapsedMonths >= tokenGrant.vestingDuration) {
        uint256 remainingGrant = tokenGrant.amount.sub(tokenGrant.totalCl…
  aimed);
        return (tokenGrant.vestingDuration, remainingGrant); // INCORRECT
    } else {
        uint16 monthsVested = uint16(elapsedMonths.sub(tokenGrant.monthsC…
  laimed));
        uint256 amountVestedPerMonth = tokenGrant.amount.div(uint256(toke…
  nGrant.vestingDuration));
        uint256 amountVested = uint256(monthsVested.mul(amountVestedPerMo…
  nth));
        return (monthsVested, amountVested);
    }
}
```

RECOMMENDATION

The code can be updated as below.

```
function calculateGrantClaim(address _recipient) private view returns (ui…
  nt16, uint256) {
    // ...
    // If over vesting duration, all tokens vested
    if (elapsedMonths >= tokenGrant.vestingDuration) {
        uint256 remainingGrant = tokenGrant.amount.sub(tokenGrant.totalCl…
  aimed);
        return (tokenGrant.vestingDuration - tokenGrant.monthsClaimed, re…
  mainingGrant); // FIXED
    } else {
        uint16 monthsVested = uint16(elapsedMonths.sub(tokenGrant.monthsC…
  laimed));
        uint256 amountVestedPerMonth = tokenGrant.amount.div(uint256(toke…
  nGrant.vestingDuration));
        uint256 amountVested = uint256(monthsVested.mul(amountVestedPerMo…
```

```
  nth));
        return (monthsVested, amountVested);
    }
}
```

UPDATES

- *Mar 09,2022*: This issue has been acknowledged and fixed by the The Last World team in commit 2307fef823b6782e57ec6f59ad9dd04416b10b6c.

2.3. Additional notes and recommendations

2.3.1. Unnecessary usage of SafeMath library in Solidity 0.8.0+ INFORMATIVE

Note: This is not a security problem. This issue is only provided as informational so readers can acknowledge the limitations of the chosen solution.

All SafeMath usage in the contract are for overflow checking, solidity 0.8.0+ already do that by default, the only usage of SafeMath now is to have a custom revert message which isn't the case in the auditing contracts. We suggest to use normal operators for readability and gas saving.

```
function calculateGrantClaim(address _recipient) private view returns (ui…
  nt16, uint256) {
    // ...
    // Check cliff was reached
    uint elapsedMonths = currentTime().sub(tokenGrant.startTime - 1 days)…
  .div(4 weeks);

    // If over vesting duration, all tokens vested
    if (elapsedMonths >= tokenGrant.vestingDuration) {
        uint256 remainingGrant = tokenGrant.amount.sub(tokenGrant.totalCl…
  aimed);
        return (tokenGrant.vestingDuration, remainingGrant);
    } else {
        uint16 monthsVested = uint16(elapsedMonths.sub(tokenGrant.monthsC…
  laimed));
        uint256 amountVestedPerMonth = tokenGrant.amount.div(uint256(toke…
  nGrant.vestingDuration));
        uint256 amountVested = uint256(monthsVested.mul(amountVestedPerMo…
  nth));
        return (monthsVested, amountVested);
```

```
    }
}
```

Snippet 1. Usage SafeMath operations in BEP20VestingVault.sol

The given source code also make use of normal math operations (integer operations without SafeMath usage). Mixed usage of SafeMath and non-SafeMath in a single codebase will confuse unaware programmer/reader as normal operations are not reverted on overflow for old solidity compiler, the overflow may be intended or not, but they will all be reverted in solidity 0.8.0+.

UPDATES

- *Mar 09,2022*: This issue has been acknowledged and fixed by the The Last World team in commit 2307fef823b6782e57ec6f59ad9dd04416b10b6c.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Mar 03, 2022*	Private Report	Verichains Lab
1.1	*Mar 09, 2022*	Public Report	Verichains Lab

Table 2. Report versions history