SECURITY AUDIT OF

CYBERIUM SMART CONTRACTS

Public Report

Jun 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 Apr 08, 2022. We would like to thank the Cyberium 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 Cyberium 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. Cyberium team has resolved and updated most of the issues following our recommendations.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Cyberium Smart Contracts

CYBERIUM is built by CYBER8 Company with a team of 25+ metaverse enthusiasts, determined to develop a true meaning of “sport metaverse”, to serve their global audiences, who are not only crypto users, but also traditional esport gamers.

CYBERIUM is not just a gameplay, it’s a platform with ultimate goal of offering esport lovers thorough - immersive experience in their sport metaverse. Once entering Cyberium, users are offered fantastic journey with different roleplays:

PLAYERS: enjoy all activities in CYBERIUM which is exclusively built for them: From main games, mini challenges on the streets, combined with classy tournament with most wanted big rewards.

CREATORS: they are not only sport lovers, but also creators who initiate amazing masterpieces, this is their world, to make their talents known, to electrify their invention ability, to express their own identity, and to bring them relative income with all of the contributions.

SOCIALIZER: CYBERIUM commits to bring a fantastic world for all users to share their passion, to connect, to meet up and to develop together in the future. There is not only gameplay, but music, entertainment, shopping variations to serve as much as possible the social needs of our Cyberium’s citizens.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of Cyberium Smart Contracts. It was conducted on commit 71472ec43120b052a1b2aee5aced3c4e532f14a1 from git repository *https://github.com/CyberEight/smart-contracts/*.

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

SHA256 Sum	File
6a47c1f804882e0b2d78dcf3259962638f4553fdc9180760c2d8f3bbcbf905e5	**./vesting/Vesting.sol**
8620798f6f9d8f9036e7c7edcecb6ab3d89153f4c024f423062e6fe8e1c289e3	**./erc721/Cue.sol**
497517410d2ae939ebdaadfd3929fb16aba4d2b5dc9363604effd8d50c7d10b6	**./erc721/Box.sol**
af45be2e9696dab65934002c4164c70b2183dec2e1c28bb39de67f125f1d94a9	**./erc1155/Card.sol**
6b730c8f2997238a48079723d0b4e72a19d4186e82f3e48ee51d2c85a947552e	**./utils/ArrayLib.sol**
6107267b418243d04b9dc47b63a18da1923681bb826474e218063980aafbe0b9	**./utils/IBotProtection.sol**

61ada15734108819c7b506960b4bcedd188ff7bcb0690550f3a0f15ff0f82193	**./utils/IEnums.sol**
518f65f4b3289effbd8d536080f8407ac3b40f7c9b2e36c14f75ae6f727fbe28	**./gacha/GachaContract.sol**
6de5fe4271b3b96df2699d769734a98597d6ac9d0e0d92a9fa16e743358e88b7	**./erc20/BreakToken.sol**
2a9d242876c35468c26dd8684d6c53eaa829d2cb1766b6f4998754a7fc839d7d	**./erc20/BotProtection.sol**
8ce6be960f971a5e9af3e610cf5e5b8be5e7e0c0d02f680bb415a83619d3c3bd	**./erc20/ESPNToken.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 Cyberium 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 and bot protection contracts

There are two types of ERC20 tokens in the Cyberium token system, they are ESPN (ESPNToken.sol) and BREAK (BreakToken.sol).

$ESPN (E-Sport Nation) is the main currency of CYBERIUM, which is used mainly as a governance token, rarely used as rewards for in-game activities. And $BREAK token is mainly used to reward players in in-game activities and the marketplace. The ESPN token is protected by a bot protection contract when transferring, this contract limits the amount users can buy or sell tokens in a block.

Note that the ESPN contract inherits the @Ownable contract and the contract owner can mint tokens if they want but the number of tokens must not exceed the configured capacity. Moreover, the contract owner can pause ESPN token transfers as well as disable the bot protection feature.

2.1.2. NFTs and marketplace-related contracts

In the audit scope, there are two types of NFT tokens, they are Cue and Box (which is used to open Cue). Their basic functions and logic are defined in Cue.sol and Box.sol contracts. Users can buy boxes via the GachaContract.sol, and they can open a Box to get a Cue via this contract also.

2.1.3. Token vesting contract

The logic for the token vesting contract is defined in Vesting.sol, this contract implements a vesting mechanism to lock and release tokens according to a configured schedule defined by the contract operator. 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, all the tokens will be unlocked at the beginning of each cliff period.

However, the owner can withdraw tokens in the vesting contract or disable vesting at any time in case of an emergency situation.

Note: Most of the contracts in the audit scope are upgradable except for the ESPNToken and Vesting contracts, which means that the contract owner can change the contract logic at any time.

2.2. Findings

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

Cyberium fixed the code, according to Verichains's draft report, in commit cf11eb9d544a1b705196587aad01f71f2ad41502.

2.2.1. Vesting.sol - Wrong formula for depositAmounts CRITICAL

In the addVestingInformationMultiple function of the Vesting contract, the depositAmounts variable is miscalculated. The depositAmounts[i] should be equal to totalVestingAmounts[i] - vestedAmounts[i].

```
function addVestingInformationMultiple(
    uint256[] memory schemeIds,
    address[] memory wallets,
    uint256[] memory startTimes,
    uint256[] memory totalVestingAmounts,
    uint256[] memory vestedAmounts,
    uint256 depositAmount
) external onlyOperator {
    // ...
    uint256[] memory depositAmounts = new uint256[](
        totalVestingAmounts.length
    );
    if (depositAmount > 0) {
        uint256 totalDepositAmount;
        for (uint256 i = 0; i < totalVestingAmounts.length; i++) {
            depositAmounts[i] = totalVestingAmounts[i] + vestedAmounts[i]…
; // WRONG FORMULA
            totalDepositAmount += depositAmounts[i];
        }
        require(
            depositAmount == totalDepositAmount,
            "Deposit amount must be equal to remaining vesting amount"
        );
    }
```

```
    // ...
}
```

RECOMMENDATION

The code should be updated as below:

```
function addVestingInformationMultiple(
    uint256[] memory schemeIds,
    address[] memory wallets,
    uint256[] memory startTimes,
    uint256[] memory totalVestingAmounts,
    uint256[] memory vestedAmounts,
    uint256 depositAmount
) external onlyOperator {
    // ...
    uint256[] memory depositAmounts = new uint256[](
        totalVestingAmounts.length
    );
    if (depositAmount > 0) {
        uint256 totalDepositAmount;
        for (uint256 i = 0; i < totalVestingAmounts.length; i++) {
            depositAmounts[i] = totalVestingAmounts[i] - vestedAmounts[i]…
; // FIXED
            totalDepositAmount += depositAmounts[i];
        }
        require(
            depositAmount == totalDepositAmount,
            "Deposit amount must be equal to remaining vesting amount"
        );
    }
    // ...
}
```

UPDATES

- *Mar 24, 2022*: This issue has been acknowledged and fixed by the Cyberium team.

2.2.2. GachaContract.sol - Contract blocking in buyGachaBox can be circumvented CRITICAL

The buyGachaBox function in the GachaContract is using isContract() function to check if the caller is a contract. However, this function only checks the code size of the caller account so that it can be bypassed by calling from a constructing contract (calling from the constructor function).

```
function buyGachaBox(
    uint256 eventId,
    address tokenCurrency,
    uint256 typeBox
) external payable whenNotPaused gachaEventExist(eventId) nonReentrant {
    require(!_msgSender().isContract(), "Caller is invalid");
    require(feeReceiver != address(0), "Not set fee receiver");
    // ...
}
```

RECOMMENDATION

A better way to prevent contracts from calling is using tx.origin, and the code can be updated as below:

```
function buyGachaBox(
    uint256 eventId,
    address tokenCurrency,
    uint256 typeBox
) external payable whenNotPaused gachaEventExist(eventId) nonReentrant {
    require(_msgSender() == tx.origin, "Caller is invalid");
    require(feeReceiver != address(0), "Not set fee receiver");
    // ...
}
```

UPDATES

- *Mar 24, 2022*: This issue has been acknowledged and fixed by the Cyberium team.

2.2.3. GachaContract.sol - Unsafe random function HIGH

The _random function in the GachaContract which is used to generate random numbers is unsafe. The safety of this function is based on some of these parameters:

- block.timestamp and block.coinbase: easy to predict in the BSC chain.
- block.difficulty: mostly unchanged (BSC)
- block.gaslimit: mostly unchanged (BSC)
- _msgSender(): mostly unchanged (BSC)
- blockhash(block.number): zero-filled bytes32 string, since the block hash of this block cannot be calculated now
- gasleft(): mostly unchanged

- _seedRandomNumber: predictable also, since it is calculated from a linear counter

```
function _random(uint256 _typeQuantity, uint256 _seedRandomNumber)
    private
    view
    returns (uint256)
{
    uint256 seed = uint256(
        keccak256(
            abi.encodePacked(
                block.timestamp +
                    block.difficulty +
                    ((
                        uint256(keccak256(abi.encodePacked(block.coinbase…
  )))
                    ) / (block.timestamp)) +
                    block.gaslimit +
                    ((uint256(keccak256(abi.encodePacked(_msgSender()))))…
   /
                        (block.timestamp)) +
                    block.number +
                    uint256(
                        keccak256(abi.encodePacked(blockhash(block.number…
  )))
                    ) +
                    gasleft() +
                    _seedRandomNumber
            )
        )
    );
    return (seed - ((seed / _typeQuantity) * _typeQuantity));
}
```

RECOMMENDATION

If the project is being deployed to the BSC chain, we can still use blockhash(block.number - 1) as a random factor. Since the block generation rate in the BSC chain is quite high, we can rest assured that this random factor will be safe enough for our purpose. A safer way to generate random numbers is using the Chainlink VRF. More detail can be found here *https://docs.chain.link/docs/chainlink-vrf/*.

UPDATES

- *Mar 24, 2022*: This issue has been acknowledged and fixed by the Cyberium team.

2.2.4. GachaContract.sol - Missing startTime check in buyGachaBox MEDIUM

In the buyGachaBox function, when user buy a gacha box, the block.timestamp is checked to ensure that it's not greater than gachaEvent.endTime. However, the gachaEvent.startTime is not checked here.

```
function buyGachaBox(
    uint256 eventId,
    address tokenCurrency,
    uint256 typeBox
) external payable whenNotPaused gachaEventExist(eventId) nonReentrant {
    require(!_msgSender().isContract(), "Caller is invalid");
    require(feeReceiver != address(0), "Not set fee receiver");
    require(
        _isWhitelistCurrency(tokenCurrency),
        "Token currency is not whitelist"
    );
    GachaEvent storage gachaEvent = _gachaEvents[eventId];
    (bool isExist, ) = ArrayLib.checkExists(gachaEvent.typeBoxes, typeBox…
  );
    require(isExist, "Type box is not exist in event");
    require(block.timestamp <= gachaEvent.endTime, "Event has ended"); //…
  MISSING startTime CHECK
    require(
        gachaEvent.status == GachaEventStatus.ACTIVE,
        "Event is not active"
    );
    // ...
}
```

RECOMMENDATION

We can update the code as below:

```
function buyGachaBox(
    uint256 eventId,
    address tokenCurrency,
    uint256 typeBox
) external payable whenNotPaused gachaEventExist(eventId) nonReentrant {
    require(!_msgSender().isContract(), "Caller is invalid");
    require(feeReceiver != address(0), "Not set fee receiver");
```

```
    require(
        _isWhitelistCurrency(tokenCurrency),
        "Token currency is not whitelist"
    );
    GachaEvent storage gachaEvent = _gachaEvents[eventId];
    (bool isExist, ) = ArrayLib.checkExists(gachaEvent.typeBoxes, typeBox…
);
    require(isExist, "Type box is not exist in event");
    require(block.timestamp >= gachaEvent.startTime, "Event hasn't starte…
d yet"); // UPDATED
    require(block.timestamp <= gachaEvent.endTime, "Event has ended");
    require(
        gachaEvent.status == GachaEventStatus.ACTIVE,
        "Event is not active"
    );
    // ...
}
```

UPDATES

- *Mar 24, 2022*: This issue has been acknowledged and fixed by the Cyberium team.

2.2.5. Vesting.sol - durationTime must be divisible by periodTime MEDIUM

When adding a new vesting by using addVestingInformation function, the vesting.durationTime should be divisible by vesting.periodTime. If this condition is not met, the vestingCount_ will be round down so that the value of vesting.periodVestingAmount will be higher than it should be. This leads to the consequence that users can claim more tokens in each period.

```
function addVestingInformation(
    uint256 schemeId,
    address wallet,
    uint256 startTime,
    uint256 totalVestingAmount,
    uint256 vestedAmount,
    uint256 depositAmount
) public onlyOperator schemeExist(schemeId) {
    require(_schemes[schemeId].isActive, "Scheme is not active");
    require(
        totalVestingAmount > 0,
        "Total vesting amount must be greater than zero"
    );
    if (startTime > 0) {
```

```
        require(startTime >= tge, "Start time must be greater than TGE");…

    }
    require(
        vestedAmount < totalVestingAmount,
        "Vested amount must be less than total vesting amount"
    );

    Scheme memory scheme = _schemes[schemeId];
    vestingCount.increment();
    uint256 vestingId = vestingCount.current();
    _vestingParticipants[wallet].push(vestingId);

    VestingInformation storage vesting = _vestings[vestingId];
    vesting.schemeId = schemeId;
    vesting.wallet = wallet;
    vesting.cliffTime = scheme.cliffTime;
    vesting.startTime = startTime == 0
        ? scheme.startTime
        : startTime + vesting.cliffTime;
    vesting.durationTime = scheme.durationTime;
    vesting.endTime = vesting.startTime + vesting.durationTime;
    vesting.periodTime = scheme.periodTime;
    vesting.totalVestingAmount = totalVestingAmount;
    uint256 vestingCount_ = vesting.durationTime / vesting.periodTime; //…
  MISSING CHECK
    vesting.periodVestingAmount = totalVestingAmount / vestingCount_;
    vesting.vestedAmount = vestedAmount;
    vesting.isActive = true;
    // ...
}
```

RECOMMENDATION

The statement require(vesting.durationTime % vesting.periodTime == 0, "...") should be added to the addVestingInformation function.

UPDATES

- *Mar 24, 2022*: This issue has been acknowledged and fixed by the Cyberium team.

2.2.6. Cue.sol, Box.sol - Storage data of the token should be deleted when burning LOW

In the Cue and Box contracts, when a token is minted, it will have corresponding storage data attached with its token id. However, when the token is burnt, its storage data is not deleted. Consider the following code in the Box contract, the _boxOfTokenId[tokenId] data should be deleted in the burn function.

```
function mint(
    address to,
    uint256 typeBox,
    bytes32 boxData
) public onlyOperator whenNotPaused returns (uint256) {
    uint256 tokenId = _tokenIdCounter.current();
    _tokenIdCounter.increment();
    _safeMint(to, tokenId);
    Box storage box = _boxOfTokenId[tokenId];
    box.typeBox = typeBox;
    box.boxData = boxData;
    emit Mint(to, tokenId, typeBox, boxData);
    return tokenId;
}

function burn(uint256 tokenId)
    public
    virtual
    override
    whenNotPaused
    onlyOperator
{
    _burn(tokenId); // _boxOfTokenId[tokenId] should be deleted
}
```

RECOMMENDATION

We should update the code as below (the fix is similar for the Cue contract):

```
function mint(
    address to,
    uint256 typeBox,
    bytes32 boxData
) public onlyOperator whenNotPaused returns (uint256) {
    uint256 tokenId = _tokenIdCounter.current();
    _tokenIdCounter.increment();
```

```
    _safeMint(to, tokenId);
    Box storage box = _boxOfTokenId[tokenId];
    box.typeBox = typeBox;
    box.boxData = boxData;
    emit Mint(to, tokenId, typeBox, boxData);
    return tokenId;
}

function burn(uint256 tokenId)
    public
    virtual
    override
    whenNotPaused
    onlyOperator
{
    _burn(tokenId);
    delete _boxOfTokenId[tokenId];
}
```

UPDATES

- *Mar 24, 2022*: This issue has been acknowledged by the Cyberium team.

2.2.7. ArrayLib.sol - Wrong index upper bound check in array LOW

All the remove functions in the ArrayLib contract have the wrong upper bound index check (off-by-one bug). These functions are:

- remove(address[] storage list, uint256 index)
- remove(uint256[] storage list, uint256 index)
- remove(string[] storage list, uint256 index)
- remove(bytes32[] storage list, uint256 index)
- removeUnchangedPosition(address[] storage list, uint256 index)
- removeUnchangedPosition(uint256[] storage list, uint256 index)
- removeUnchangedPosition(string[] storage list, uint256 index)
- removeUnchangedPosition(bytes32[] storage list, uint256 index)

Below is a sample code snippet that have wrong upper bound index check:

```
function remove(bytes32[] storage list, uint256 index) internal {
    require(
        list.length > 0 && index <= list.length, // INCORRECT
        "Array Library: List and/or index is invalid"
    );
```

```
    list[index] = list[list.length - 1];
    list.pop();
}
```

RECOMMENDATION

The above function can be fixed as below:

```
function remove(bytes32[] storage list, uint256 index) internal {
    require(
        list.length > 0 && index < list.length, // FIXED
        "Array Library: List and/or index is invalid"
    );
    list[index] = list[list.length - 1];
    list.pop();
}
```

UPDATES

- *Mar 24, 2022*: This issue has been acknowledged and fixed by the Cyberium team.

2.3. Additional notes and recommendations

2.3.1. Vesting.sol - Wrong function name getVestingParticipants INFORMATIVE

The name of the getVestingParticipants function of the Vesting contract is wrong and misleading. This function returns all the vestings in which the user with the address wallet is currently participating.

```
function getVestingParticipants(address wallet)
    external
    view
    returns (uint256[] memory)
{
    return _vestingParticipants[wallet];
}
```

RECOMMENDATION

The name of this function should be changed to getParticipatedVestings.

UPDATES

- *Mar 24, 2022*: This issue has been acknowledged and fixed by the Cyberium team.

2.3.2. Vesting.sol - availableAmount should only be calculated when the vesting is claimable INFORMATIVE

In the _claim function of the Vesting contract, the statement which calculates the availableAmount should not be run if the vesting is not claimable.

```
function _claim(
    uint256[] memory _vestingIds,
    uint256 _totalAvailableVestingIds
) internal {
    uint256 index = 0;
    uint256 totalClaimableAmount = 0;

    uint256[] memory vestingIdsAvailale = new uint256[](
        _totalAvailableVestingIds
    );
    uint256[] memory claimableAmounts = new uint256[](
        _totalAvailableVestingIds
    );

    for (uint256 i = 0; i < _vestingIds.length; i++) {
        uint256 availableAmount = _getAvailableAmount(_vestingIds[i]); //…
  THIS STATEMENT SHOULDN'T BE HERE
        if (_isClaimableVesting(_vestingIds[i])) {
            vestingIdsAvailale[index] = _vestingIds[i];
            claimableAmounts[index] = availableAmount;
            totalClaimableAmount += availableAmount;
            index++;

            VestingInformation storage vesting = _vestings[_vestingIds[i]…
  ];
            vesting.vestedAmount += availableAmount;
        }
    }
    // ...
}
```

RECOMMENDATION

The code can be updated as below:

```
function _claim(
    uint256[] memory _vestingIds,
```

```
    uint256 _totalAvailableVestingIds
) internal {
    uint256 index = 0;
    uint256 totalClaimableAmount = 0;

    uint256[] memory vestingIdsAvailale = new uint256[](
        _totalAvailableVestingIds
    );
    uint256[] memory claimableAmounts = new uint256[](
        _totalAvailableVestingIds
    );

    for (uint256 i = 0; i < _vestingIds.length; i++) {
        if (_isClaimableVesting(_vestingIds[i])) {
            uint256 availableAmount = _getAvailableAmount(_vestingIds[i])…
  ; // FIXED
            vestingIdsAvailale[index] = _vestingIds[i];
            claimableAmounts[index] = availableAmount;
            totalClaimableAmount += availableAmount;
            index++;

            VestingInformation storage vesting = _vestings[_vestingIds[i]…
  ];
            vesting.vestedAmount += availableAmount;
        }
    }
    // ...
}
```

UPDATES

- *Mar 24, 2022*: This issue has been acknowledged and fixed by the Cyberium team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Mar 24, 2022*	Public Report	Verichains Lab
1.1	*Apr 08, 2022*	Public Report	Verichains Lab
1.2	*Jun 03, 2022*	Public Report	Verichains Lab
1.3	*Jun 09, 2022*	Public Report	Verichains Lab

Table 2. Report versions history