BlockSec

Security Audit Report for Multichain veMULTI Contracts

Date: Apr 22, 2022

Version: 1.0

Contact: contact@blocksecteam.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.04.24 17:55:51 +08'00'

Contents

Report Manifest

Item	Description
Client	Multichain
Target	Multichain veMULTI Contracts

Version History

Version	Date	Description
1.0	Apr 22, 2022	First Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values of the repo [1] during the audit are shown in the following.

Project		Commit SHA
veMULTI	Version 1	bac804399d1ea280e5bd8cdc9488b6fa6a0a7fcc
	Version 2	6c6e267aaca71dd9e4b9f63bfab9a855d9638e2a

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report do not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team).

[1] https://github.com/anyswap/veMULTI

We also manually analyze possible attack scenarios with independent auditors to cross-check the result.

- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.
- **Fixed** The issue has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **nine** potential issues. We have **five** recommendations.

- High Risk: 0
- Medium Risk: 3
- Low Risk: 6
- Recommendations: 5

ID	Severity	Description	Category	Status
1	Medium	Unchecked ERC-721 Callback Result	Software Security	Fixed
2	Low	Improper Check for the Return Values of the `transferFrom` Function	Software Security	Acknowledged
3	Medium	Incorrect Address Used in the `_burn` Function	Software Security	Fixed
4	Low	Access Out Of Bounds in the `getBlockByTime` Function	Software Security	Acknowledged
5	Low	Unchecked Arrays in the `claimRewardMany` Function	Software Security	Fixed
6	Low	Inconsistent Implementation of the Burn Logic	Software Security	Fixed
7	Medium	Inconsistent Handling of Epoch Time	DeFi Security	Fixed
8	Low	Inconsistent End Time in the `addEpochBatch` Function	DeFi Security	Fixed
9	Low	Inconsistent Implementation of the Reward Calculation	DeFi Security	Acknowledged
10	-	Check Zero Address In the `ve.ownerOf` Function	Recommendation	Fixed
11	-	Implement Secure Logic for the `transferAdmin` Function	Recommendation	Fixed
12	-	Avoid Continuous Divisions in the `_pendingRewardSingle` Function	Recommendation	Fixed
13	-	Alleviate the Concern of Potential Centrality	Recommendation	Acknowledged
14	-	Follow the Checks-Effects-Interactions Pattern	Recommendation	Fixed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Unchecked ERC-721 Callback Result

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `ve` contract is forked from the `VotingEscrow` contract of the Curve project. The original account-based `VotingEscrow` contract is modified into an ERC-721 based NFT token contract. In the implementation of the `safeTransferFrom` function, according to the ERC-721 standard, the caller should check the return value of the callback, and revert if the returned signature is incorrect. This check is not implemented in the `safeTransferFrom` function. As a result, tokens transferred to a contract without proper ERC-721 token support would be locked.

```
function safeTransferFrom(
    address _from,
    address _to,
    uint _tokenId,
    bytes memory _data
) public {
    _transferFrom(_from, _to, _tokenId, msg.sender);

    if (_isContract(_to)) {
        // Throws if transfer destination is a contract which does not implement 'onERC721Received'
        try IERC721Receiver(_to).onERC721Received(msg.sender, _from, _tokenId, _data) returns (bytes4) {} catch (
            bytes memory reason
        ) {
            if (reason.length == 0) {
                revert('ERC721: transfer to non ERC721Receiver implementer');
            } else {
                assembly {
                    revert(add(32, reason), mload(reason))
                }
            }
        }
    }
}
```

Listing 2.1: ve.sol

Impact Tokens transferred to a contract without proper ERC-721 token support would be locked.

Suggestion Check the return value of the callback in the `safeTransferFrom` function.

2.1.2 Improper Check for the Return Values of the `transferFrom` Function

Severity Low

Status Acknowledged

Introduced by Version 1

Description The `_deposit_for` function in the `ve` contract transfers the deposit token to the contract and checks the result of this transfer. This can fail due to improperly implemented tokens (for example, the USDT token) that does not return value in the `transferFrom` function.

```
address from = msg.sender;
if (_value != 0 && deposit_type != DepositType.MERGE_TYPE) {
    assert(IERC20(token).transferFrom(from, address(this), _value));
}
```

Listing 2.2: ve.sol

Impact If non-standard token is used as the deposit token in the `ve` contract, deposits may fail due to this check.

Suggestion Use common libraries like `SafeERC20` of OpenZeppelin.

Feedback from the Project The deposit token for this contract is fixed (i.e. the `MULTI` token, with address `0x65ef703f5594d2573eb71aaf55bc0cb548492df4`). This token has a standard `transferFrom` function.

2.1.3 Incorrect Address Used in the `_burn` Function

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the ERC-721 logic implementation of the `ve` contract, the `_burn` function calls the `_removeTokenFrom` function in the end to remove the token ID from its owner. However, it is implemented incorrectly to remove the token ID from the `msg.sender`, not its real owner.

```
function _burn(uint _tokenId) internal {
    require(_isApprovedOrOwner(msg.sender, _tokenId), "caller is not owner nor approved");

    address owner = ownerOf(_tokenId);

    // Clear approval
    approve(address(0), _tokenId);
    // Remove token
    _removeTokenFrom(msg.sender, _tokenId);
    emit Transfer(owner, address(0), _tokenId);
}
```

Listing 2.3: ve.sol

Impact If the operator (rather than the real owner) calls this function, it may fail because the operator does not own the token ID.

Suggestion Change the first parameter of the `_removeTokenFrom` call to `owner`.

2.1.4 Access Out Of Bounds in the `getBlockByTime` Function

Severity Low

Status Acknowledged

Introduced by `Version 1`

Description The `VEReward` contract is used to distribute rewards to the holders of the NFTs issued by the `ve` contract. In this contract, the `getBlockByTime` function is used to estimate the block number of any time, given the `point_history` records in this contract. However, this function is not implemented properly. For example, if the target `_time` parameter exceeds all historical records in the `point_history`, an access out of bounds can occur due to the improper handling of the edge cases.

```
function getBlockByTime(uint _time) public view returns (uint) {
    // Binary search
    uint _min = 0;
    uint _max = point_history.length - 1; // asserting length >= 2
    for (uint i = 0; i < 128; ++i) {
        // Will be always enough for 128-bit numbers
```

```
        if (_min >= _max) {
            break;
        }
        uint _mid = (_min + _max + 1) / 2;
        if (point_history[_mid].ts <= _time) {
            _min = _mid;
        } else {
            _max = _mid - 1;
        }
    }

    Point memory point0 = point_history[_min];
    Point memory point1 = point_history[_min + 1];
    // asserting point0.blk < point1.blk, point0.ts < point1.ts
    uint block_slope; // dblock/dt
    block_slope = (BlockMultiplier * (point1.blk - point0.blk)) / (point1.ts - point0.ts);
    uint dblock = (block_slope * (_time - point0.ts)) / BlockMultiplier;
    return point0.blk + dblock;
}
```

Listing 2.4: VEReward.sol

Impact If the `_time` parameter exceeds all historical records in the `point_history`, this function will revert.

Suggestion Revise the code logic to properly handle edge cases of the binary search.

Feedback from the Project The `getBlockByTime` function is only used in the `claimReward` function. If the `_time` parameter is correctly passed, it won't cause any problems. If the `_time` parameter is wrongly passed, the function will revert and thus causes no problems. Besides, from the point of gas consumption, addtional checks are not desired.

2.1.5 Unchecked Arrays in the `claimRewardMany` Function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `VEReward` contract provides a `claimRewardMany` function so that users can withdraw rewards in many epochs for many token IDs. However, the function does not check the lengths of the parameters.

```
function claimRewardMany(uint[] calldata tokenIds, Interval[][] calldata intervals) public
     returns (uint[] memory rewards) {
    rewards = new uint[] (tokenIds.length);
    for (uint i = 0; i < tokenIds.length; i++) {
        rewards[i] = claimReward(tokenIds[i], intervals[i]);
    }
    return rewards;
}
```

Listing 2.5: VEReward.sol

Impact N/A

Suggestion Check the lengths of the array parameters.

2.1.6 Inconsistent Implementation of the Burn Logic

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description The ve contract has a custom implementation of the ERC-721 standard. The _burn function implemented in this contract is inconsistent with the standard OpenZeppelin's implementation. Specifically, the approve function forbids the approved account to call the approve function. However, in the original OpenZeppelin's implementation, the access control for _burn is _isApprovedOrOwner, which is comprised of the following three conditions: spender == owner, isApprovedForAll(owner, spender), or getApproved(tokenId).

```
function _burn(uint _tokenId) internal {
    require(_isApprovedOrOwner(msg.sender, _tokenId), "caller is not owner nor approved");

    address owner = ownerOf(_tokenId);

    // Clear approval
    approve(address(0), _tokenId);
    // Remove token
    _removeTokenFrom(msg.sender, _tokenId);
    emit Transfer(owner, address(0), _tokenId);
}
```

Listing 2.6: ve.sol

```
function approve(address _approved, uint _tokenId) public {
    address owner = idToOwner[_tokenId];
    // Throws if '_tokenId' is not a valid NFT
    require(owner != address(0));
    // Throws if '_approved' is the current owner
    require(_approved != owner);
    // Check requirements
    bool senderIsOwner = (idToOwner[_tokenId] == msg.sender);
    bool senderIsApprovedForAll = (ownerToOperators[owner])[msg.sender];
    require(senderIsOwner || senderIsApprovedForAll);
    // Set the approval
    idToApprovals[_tokenId] = _approved;
    emit Approval(owner, _approved, _tokenId);
}
```

Listing 2.7: ve.sol

```
function _burn(uint256 tokenId) internal virtual {
    address owner = ERC721.ownerOf(tokenId);

    _beforeTokenTransfer(owner, address(0), tokenId);
```

```

        // Clear approvals
        _approve(address(0), tokenId);

        _balances[owner] -= 1;
        delete _owners[tokenId];

        emit Transfer(owner, address(0), tokenId);

        _afterTokenTransfer(owner, address(0), tokenId);
    }
```

Listing 2.8: openzeppelin-contracts/ERC721.sol

```
    function _isApprovedOrOwner(address spender, uint256 tokenId) internal view virtual returns (
        bool) {
        require(_exists(tokenId), "ERC721: operator query for nonexistent token");
        address owner = ERC721.ownerOf(tokenId);
        return (spender == owner || isApprovedForAll(owner, spender) || getApproved(tokenId) ==
            spender);
    }
```

Listing 2.9: openzeppelin-contracts/ERC721.sol

```
    function burn(uint256 tokenId) public virtual {
        //solhint-disable-next-line max-line-length
        require(_isApprovedOrOwner(_msgSender(), tokenId), "ERC721Burnable: caller is not owner nor
            approved");
        _burn(tokenId);
    }
```

Listing 2.10: openzeppelin-contracts/ERC721Burnable.sol

Impact N/A

Suggestion Revise the code to keep compatible with the standard implementation.

2.2 DeFi Security

2.2.1 Inconsistent Handling of Epoch Time

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `VEReward` contract provides token rewards to NFTs created in the `ve` contract. The rewards are distributed in different epochs, while the project admin is allowed to create epochs, setting the start time, end time and total rewards in this epoch. Two functions, i.e., the `addEpoch` function for creating a single epoch and the `addEpochBatch` function for creating several continuous epochs, are implemented to serve the purpose. However, these two functions are not consistent. Specifically, the `addEpoch` function checks the end time of the epoch to ensure that the end time has not passed (i.e., less than block.timestamp). In contrast, the addEpochBatch function does not check the end time for intermediate epochs, which suggests that creating an epoch that has passed is allowed.

```
function addEpochBatch(uint startTime, uint endTime, uint epochLength, uint totalReward)
     external onlyAdmin returns(uint, uint, uint) {
    assert(block.timestamp < endTime && startTime < endTime);
    if (epochInfo.length > 0) {
        require(epochInfo[epochInfo.length - 1].endTime <= startTime);
    }
    uint numberOfEpoch = (endTime + 1 - startTime) / epochLength;
    uint _reward = totalReward / numberOfEpoch;
    uint _start = startTime;
    uint _end;
    uint _epochId;
    uint accurateTR;
    for (uint i = 0; i < numberOfEpoch; i++) {
        _end = _start + epochLength;
        (_epochId, accurateTR) = _addEpoch(_start, _end, _reward);
        _start = _end;
    }
    uint lastPointTime = point_history[point_history.length - 1].ts;
    if (lastPointTime < block.timestamp) {
        addCheckpoint();
    }
    emit LogAddEpoch(startTime, _end, epochLength, _epochId + 1 - numberOfEpoch);
    return (_epochId + 1 - numberOfEpoch, _epochId, accurateTR * numberOfEpoch);
}
```

Listing 2.11: VEReward.sol

Impact It is possible to create epochs whose end time have passed.

Suggestion Add sanity checks to maintain the consistency of the epoch creation logic.

2.2.2 Inconsistent End Time in the addEpochBatch Function

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description The implementation of the addEpochBatch function may cause the inconsistency between the calculated end time of the final period and the endTime parameter. For example, there will be only one epoch if epochLength > (endTime - startTime) / 2. In such a case, the calculated end time is different from the endTime passed into the function.

```
function addEpochBatch(uint startTime, uint endTime, uint epochLength, uint totalReward)
     external onlyAdmin returns(uint, uint, uint) {
    assert(block.timestamp < endTime && startTime < endTime);
    if (epochInfo.length > 0) {
        require(epochInfo[epochInfo.length - 1].endTime <= startTime);
    }
    uint numberOfEpoch = (endTime + 1 - startTime) / epochLength;
    uint _reward = totalReward / numberOfEpoch;
```

```
        uint _start = startTime;
        uint _end;
        uint _epochId;
        uint accurateTR;
        for (uint i = 0; i < numberOfEpoch; i++) {
            _end = _start + epochLength;
            (_epochId, accurateTR) = _addEpoch(_start, _end, _reward);
            _start = _end;
        }
        uint lastPointTime = point_history[point_history.length - 1].ts;
        if (lastPointTime < block.timestamp) {
            addCheckpoint();
        }
        emit LogAddEpoch(startTime, _end, epochLength, _epochId + 1 - numberOfEpoch);
        return (_epochId + 1 - numberOfEpoch, _epochId, accurateTR * numberOfEpoch);
    }
```

Listing 2.12: VEReward.sol

Impact N/A

Suggestion Revise the batch epoch creation logic.

2.2.3 Inconsistent Implementation of the Reward Calculation

Severity Low

Status Acknowledged

Introduced by `Version 1`

Description There exists an inconsistency in the reward calculation logic in the `VEReward` contract. To distribute the reward, the `claimReward` function will invoke the `_pendingRewardSingle` function to calculate the reward amount. However, there is another function named `pendingReward` (which is a `view` function) with slightly different logic for the reward calculation.

Impact N/A

Suggestion Maintain the consistency of the reward calculation logic.

Feedback from the Project The actual reward calculation is in the `_pendingRewardSingle` function. The `pendingReward` is only a `view` function for front-end display. If there are any differences, the result returned by the `_pendingRewardSingle` function is used.

2.3 Additional Recommendation

2.3.1 Check Zero Address In the `ve.ownerOf` Function

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the standard OpenZeppelin's implementation of the ERC-721 token, the `ownerOf` function will check whether the owner address is zero, to prevent returning owners of non-existent tokens (which is a zero address). It is recommended to implemented the same logic in the `ve` contract.

```
function ownerOf(uint _tokenId) public view returns (address) {
    return idToOwner[_tokenId];
}
```

Listing 2.13: ve.sol

```
function ownerOf(uint256 tokenId) public view virtual override returns (address) {
    address owner = _owners[tokenId];
    require(owner != address(0), "ERC721: owner query for nonexistent token");
    return owner;
}
```

Listing 2.14: openzeppelin-contracts/ERC721.sol

Impact N/A

Suggestion Add the corresponding sanity checks.

2.3.2 Implement Secure Logic for the `transferAdmin` Function

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The current implementation of the `transferAdmin` function directly changes the admin address. It is suggested that the admin transfer follows the *Transfer-Accept* pattern as used in the Compound project. Specifically, the `transferAdmin` function should only change the pending admin to be set, and another function named `acceptAdmin` is used to set the pending admin to the actual admin.

```
function transferAdmin(address _admin) external onlyAdmin {
    admin = _admin;
}
```

Listing 2.15: ve.sol

Impact N/A

Suggestion Implement secure logic for the transfer admin procedure.

2.3.3 Avoid Continuous Divisions in the `_pendingRewardSingle` Function

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The current implementation of the `_pendingRewardSingle` function has continuous divisions. It is recommended to refactor the original logic into a multiplication to prevent the potential precision loss.

```
    uint reward = epoch.rewardPerSecond * (end - last) * power / epoch.totalPower /
        RewardMultiplier;
    return (reward, finished);
}
```

Listing 2.16: VEReward.sol

Impact Potential precision loss.

Suggestion Remove the continuous divisions.

2.3.4 Alleviate the Concern of Potential Centrality

Status Acknowledged

Introduced by `Version 1`

Description The reward token in the `VEReward` contract is directly transferred from the project admin to the contract. There is no explicit logic that requires the project admin to transfer reward token to the contract when adding an epoch through invoking the `addEpoch` function (or the `addEpochBatch` function). Therefore, it is not guaranteed (as in the contract level) that users will always be able to fully withdraw the rewards. This is subject to the centrality problem.

Impact N/A

Suggestion Transfer reward in the `addEpoch` function.

Feedback from the Project Because the exact reward amount cannot be accurately calculated and someone may not claim the rewards for any reasons, it will be directly set by the project admin, and in the not soon future, this admin will be transfer to MultiDAO.

2.3.5 Follow the Checks-Effects-Interactions Pattern

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `withdraw` function of the `ve` contract, the implementation does not follow the Checks-Effects-Interactions pattern, i.e., the transfer is done before the NFT is burned.

```
        assert(IERC20(token).transfer(msg.sender, value));

        // Burn the NFT
        _burn(_tokenId);

        emit Withdraw(msg.sender, _tokenId, value, block.timestamp);
        emit Supply(supply_before, supply_before - value);
    }
```

Listing 2.17: ve.sol

Impact N/A

Suggestion Follow the Checks-Effects-Interactions pattern.

Feedback from the Project There is a reentrancy guard in the `withdraw` function of the `ve` contract.