#HashEx
BLOCKCHAIN SECURITY

Volcano

smart contracts preliminary audit report for internal use only

April 2024

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Volcano team to perform an audit of their smart contracts. The audit was conducted between 07/04/2024 and 10/04/2024

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code was provided directly in .sol files. The SHA-1 hashes of the audited files are:

SmartChef.sol	a4fa50201bb1cdaff4f06307eee899dea3b7a9e3
vault.sol	581823cb65b1dc7870337892ee43b4aaf7099b18

2.1 Summary

Project name	Volcano
Platform	Blast
Language	Solidity
Centralization level	High
Centralization risk	High

2.2 Contracts

Name	Address
SmartChef	
Vault	

3. Project centralization risks

Cd0CR1e Owner privileges

- The contract owner can update the vault address. The vault receives all yiedls for WETH and USDB deposited to the contract. Setting the wrong vault address will also break deposits.

- Set arbitrary big reward per block. Setting the reward to an extremely big value will devaluate the token.

- Update minimum deposit per user. Setting it the an extremely big value will block deposits.

- Update rate that goest to the Vault on withdrawals up to 0.03%.

- Turn on and off emergency withdrawals.

- Finish the reward token distribution.

Cd1CR1f Owner privileges

- The contract owner can set arbitrary big reward rates for the WETH and USDB deposits. Setting them to extremely high value will allow a user who requests rewards first to take all the rewards currently on the contract.

- The contract owner can update the farm address.

4. Found issues

High	1 (8%)
Medium	4 (31%)
Low	5 (38%)
Info	3 (23%)

Cd0. SmartChef

ID	Severity	Title	Status
Cd0I4c	High	Lack of constraints on reward per block	Open
Cd0I4e	Medium	Calling the deposit() function with zero amount will effectively burn reward tokens to user	Open
Cd0I4d	Medium	Inconsistent documentation	Open
Cd0I4f	Low	Lack of events	Open
Cd0I50	Low	Gas optimizations	Open
Cd0I52	Info	Non conventional naming	Open
Cd0I51	Info	No way to set the minimum deposit for a pool smaller	Open

Cd1. Vault

ID	Severity	Title	Status
Cd1I5d	Medium	Same amount of interest reward is distributed for users with and without an inviter	Open
Cd1I68	Medium	Rewards after SmartChef farming end	Open
Cd1I60	Low	Gas optimizations	Open
Cd1I5f	Low	Lack of events	Open
Cd1I5e	Low	Result of token transfers is not checked	Open
Cd1I63	Info	The rewards are not guaranteed	Open

5. Contracts

Cd0. SmartChef

Overview

A MasterChef-like contract for one pool. The pool can be either for WETH or USDB tokens. Users who stake in the pool receive additional rewards in the reward token. The reward token (VCN in terms of the contract) has no constant mint rate, it is minted proportionally to the amount of staked tokens.

Part of the tokens on withdrawal are sent to the Vault contract.

The contract has a minimum deposit feature. If a user deposits at once an amount of tokens bigger than the minimum amount, he is eligible for additional reward distribution from the Vault contract.

The USDB and WETH yields are distributed to the Vault contract.

Issues

Cd0I4c Lack of constraints on reward per block

High Open

The contract mints rewards token proportionally the amount of staked token. The rate can be changed by the contract owner.

```
    function updateRewardPerBlock(uint256 _rewardPerBlock) external onlyOwner {          //□
□□
        _updatePool();
        rewardPerBlock = _rewardPerBlock;
        emit NewRewardPerBlock(_rewardPerBlock);
    }
```

Setting the `rewardPerBlock` to an extremely big value means high mint rate and token

devaluation.

Recommendation

Add an upper constraint for the `_rewardPerBlock` parameter to avoid setting it to an extremely big value.

Cd0l4e Calling the deposit() function with zero amount will effectively burn reward tokens to user

Medium Open

If a user calls the `deposit()` function with zero amount or the `depositEth()` function without passing native currency the contract sets the unclaimed VCN token rewards to zero. However, it does not distribute the accumulated rewards to the user before resetting the amount. This behavior results in the loss of unclaimed rewards that the user has earned.

```
function deposit(uint256 _amount) public nonReentrant {
    ...
    if (user.amount > 0) {
        uint256 pending = user.amount * accTokenPerShare / 1e18 - user.rewardDebt;
        if (pending > 0) {
            IVCN(address(rewardToken)).mint(address(msg.sender), pending);
            rewardClaimed[msg.sender] += pending;
        }
    }
    user.rewardDebt = user.amount * accTokenPerShare / 1e18;
    ...
```

Recommendation

Update the user reward debt only if the rewards were sent to the user.

Cd0l4d Inconsistent documentation

Medium Open

The variable `rewardPerBlock` is named to suggest that it represents a constant token mint rate per block, implying a fixed number of tokens minted in each block. However, the contract

does not operate as implied by this variable's name. Instead of minting a constant amount of reward tokens per block, the contract mints an amount of VCN (the reward token) that is proportional to the total amount of stake tokens deposited in the contract.

```
    // VCN tokens created per block.
    uint256 public rewardPerBlock;
```

Recommendation

Given the lack of comprehensive documentation, it is challenging to determine if the current behavior of the `rewardPerBlock` variable is intentional.

If the Behavior is Intentional:

- Update the in-code documentation to provide a clear and detailed explanation of the variable's functionality, ensuring that it accurately reflects the proportional minting mechanism.
- Rename the variable to more accurately reflect its operational logic.

If the Behavior is Unintentional:

- Adjust the contract's logic to align with the expected constant token mint rate per block, if that is the intended functionality.

Cd0l4f Lack of events

Low Open

The functions `setVault()`, `setFeeRate()`, `setEmergencySwitch()`, `stopReward()` change important variable in the contract storage, but no event is emitted.

We recommend adding events to these functions to make it easier to track their changes offline.

Cd0I50 Gas optimizations

Low Open

- The `PRECISION_FACTOR` constant is not used.

- The `RewardsStop` event is not used.

- The NewStartAndEndBlocks event is not used.

- no need to update the rewardDebt value in `claim()` function if pending is zero.

- `startBlock`, `rewardToken`, `stakedToken` can be set immutable.

- Multiple reads from storage of the same values. Use memoization.

Cd0I52 Non conventional naming

Info Open

The constant variables `RateBase` and `blocksPerYear` do not adhere to Solidity naming conventions for constants. We recommend renaming them to uppercase with underscores to improve readability and consistency with Solidity standards. The suggested names would be `RATE_BASE` and `BLOCKS_PER_YEAR`.

Cd0I51 No way to set the minimum deposit for a pool smaller

Info Open

The smart contract lacks a mechanism to reduce the minimum deposit amount once it is set. If the minimum deposit threshold is mistakenly configured to an excessively high value, it effectively prevents any further deposits due to the inability to meet the elevated requirement.

```
    function updatePoolLimitPerUser(bool _hasUserLimit, uint256 _poolLimitPerUser)
external onlyOwner {
        require(hasUserLimit, "Must be set");
        if (_hasUserLimit) {
            require(_poolLimitPerUser > poolLimitPerUser, "New limit must be higher");
            poolLimitPerUser = _poolLimitPerUser;
```

```
        } else {
            hasUserLimit = _hasUserLimit;
            poolLimitPerUser = 0;
        }
        emit NewPoolLimit(poolLimitPerUser);
    }
```

Cd1. Vault

Overview

A vault contract that receives parts of the user deposits to the SmartChef contract and yield from USDB and WETH token deposits to the SmartChef contract.

The contract issues rewards in WETH and USDB tokens based on the amount of users' claimed rewards in the SmartChef contract. The rate of these rewards is set by the contract owner.

The contract includes a bonus distribution mechanism for the last five users who deposited an amount exceeding the minimum required in the SmartChef contract. This bonus distribution occurs no more frequently than once every three days.

The vault contract also implements a referral system for users who deposited in the SmartChef contract.

Issues

Cd1I5d Same amount of interest reward is distributed for users with and without an inviter

Medium Open

The function `claimInterestReward()` calculates and distributes the reward to the users. It calculates the reward amount separately for a user with and without an invite. However, the calculated amount is the same.

```
    function claimInterestReward(address _interestToken) external {
        require(_interestToken == WETH || _interestToken == USDB, "interest token must be
weth or usdb");
        address l1Inviter = inviteInfo[msg.sender].inviter;
        address l2Inviter = inviteInfo[l1Inviter].inviter;
        ...
        if(l1Inviter != address(0)){
            uint l1Amount = amount * distributeRate.l1InviteRate / inviteRateBase;
            userInviteReward[_interestToken][l1Inviter] += l1Amount;
            if(l2Inviter != address(0)){
                uint l2Amount = amount * distributeRate.l2InviteRate / inviteRateBase;
                userInviteReward[_interestToken][l2Inviter] += l2Amount;
            }
            uint userAmount = amount * (inviteRateBase - distributeRate.l1InviteRate -
distributeRate.l2InviteRate - distributeRate.vaultRate - distributeRate.luckyPoolRate) /
inviteRateBase;
            IERC20(_interestToken).transfer(msg.sender, userAmount);
        }else{
            uint userAmount = amount * (inviteRateBase - distributeRate.l1InviteRate -
distributeRate.l2InviteRate - distributeRate.vaultRate - distributeRate.luckyPoolRate) /
inviteRateBase;
            IERC20(_interestToken).transfer(msg.sender, userAmount);
        }
    }
```

Recommendation

A lack of documentation makes it hard to tell whether it is an intended behavior. If it is, calculate the reward amount out of the if block.

Cd1168 Rewards after SmartChef farming end

Medium Open

The WETH and USDB rewards that a user is eligible to receive are calculated based on the user's rewards in the SmartChef contract. Once the farming period in the SmartChef contract concludes, the distribution of rewards ceases. Users who have claimed all their rewards in the Vault contract will not be able to claim any subsequent rewards. However, the Vault contract can still receive rewards, but these will not be distributed to the users.

```
    function pendingInterestReward(address _interestToken, address _user) public view
returns(uint256){
        uint256 amount;
        address farm;
        if(_interestToken == WETH){
            farm = ethFarm;
        }else if(_interestToken == USDB){
            farm = usdbFarm;
        }else{
            return 0;
        }
        amount = IFarm(farm).rewardClaimed(_user) + IFarm(farm).pendingReward(_user);
        uint256 accInterestReward = amount * rewardPerBlockScale[_interestToken]; //@audit
        return accInterestReward - userInterestClaimed[_user][_interestToken];
    }
```

Cd1160 Gas optimizations

Low Open

- The `index2User` mapping is not used

- Multiple reads from storage of the same values. Use memoization.

Cd115f Lack of events

Low Open

The functions `setFarm()`, `setRewardPerBlockScale()`, `emergencyWithdraw()`change important variables in the contract storage, but no event is emitted.

We recommend adding events to these functions to make it easier to track their changes offline.

Cd115e Result of token transfers is not checked

Low Open

The contract does not check the returned results of the ERC20 transfer function.

```
            IERC20(_interestToken).transfer(msg.sender, userAmount);
```

The ERC20 token standard mandates that the token transfer function should return a boolean value indicating the success or failure of the token transfer. Typically, tokens are designed to always return true, with the transaction failing if the transfer is unsuccessful. However, it is considered best practice to robustly handle the return values to ensure reliability.

Additionally, it is important to note that some implementations of the ERC20 token do not adhere strictly to the ERC20 standard and might not return a boolean upon transfer. Consequently, to accommodate all scenarios, it is advisable to utilize a library that addresses these variations, such as OpenZeppelin's SafeERC20, which provides a more secure and standardized approach to handling ERC20 token transfers.

Cd1I63 The rewards are not guaranteed

Info Open

The amount of available rewards for users is not calculated based on the amount that the Vault received but by the rates set by the contract owner. If the contract owner sets reward rates smaller than needed, some WETH and USDB reward tokens won't be able to be claimed. If the contract owner sets a bigger reward than needed, some users may not be able to get their rewards as there won't be enough tokens on the vault contract.

6. Conclusion

1 high, 4 medium, 5 low severity issues were found during the audit. No issues were resolved in the update. The reviewed contracts are highly dependent on the owner's account. See the centralization risks chapter.

This audit includes recommendations on code improvement and the prevention of potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. Issue status description

Resolved. The issue has been completely fixed.

Partially fixed. Parts of the issue have been fixed but the issue is not completely resolved.

Acknowledged. The team has been notified of the issue, no action has been taken.

Open. The issue remains unresolved.

Appendix C. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

Appendix D. Centralization risks classification

Centralization level

- **High.** The project owners can manipulate user's funds, lock user's funds on their will (reversible or irreversible), or maliciously update contracts parameters or bytecode.
- **Medium.** The project owners can modify contract's parameters to break some functions of the project contract or contracts, but user's funds remain withdrawable.
- **Low.** The contract is trustless or its governance functions are safe against a malicious owner.

Centralization risk

- **High.** Lost ownership over the project contract or contracts may result in user's losses. Contract's ownership belongs to EOA or EOAs, and their security model is unknown or out of scope.
- **Medium.** Contract's ownership is transferred to a contract with not industry-accepted parameters, or to a contract without an audit. Also includes EOA with a documented security model, which is out of scope.
- **Low.** Contract's ownership is transferred to a well-known or audited contract with industry-accepted parameters.

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY