HashEx

Hepa Finance

smart contracts audit report

Prepared for:
hepa.finance

Authors: HashEx audit team

June 2021

Contents

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.

DISCLAIMER: 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, 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 (https://hashex.org).

Introduction

HashEx was commissioned by the Hepa Finance team to perform an audit of their smart contracts. The audit was conducted between June 23 and June 26, 2021.

The code deployed to Binance Smart Chain (BSC):
HepaToken `0xba638f51052b655380E6ea8e857f42b39344ADc7` in mainnet,
MasterHepa `0xd3260Bdec435b0E4388622DE6d16d7ef3Fcd1F9f` in testnet.
Code was provided without documentation.

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 to understand 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.

Update: Hepa Finance team has responded to this report. Individual responses were added after each item in the section. The updated code is deployed to the testnet of BSC:
HepaToken `0x422322FBD72899A26eEF3e2b3e505a2e80E78fCB`,
MasterHepa `0x88Bacad4a016026863D9A96c424041998cdF09D9`.

Contracts overview

`HepaToken.sol`

Implementation of ERC20 token standard with the locking and unlock functions under the owner's control. Initialized GovernanceToken.

`GovernanceToken.sol`

Implementation of ERC20 token standard with the locking and unlock functions under the owner's control.

`Authorizable.sol`

Extension of the standard Ownable contract. Adds the authorized addresses list controlled by the owner.

`MasterHepa.sol`

Staking contract with locking rewards mechanism.

Found issues

ID	Title	Severity	Response
01	Governance: no safety guards for lockFromBlock and lockToBlock	High	Fixed
02	Governance: canUnlockAmount() could return wrong value	High	Fixed
03	Governance: mint() is open for the owner	High	Informed
04	MasterHepa: no safety guards for devFeeStage and userFeeStage	High	Fixed
05	MasterHepa: owner controls user.lastWithdrawBlock	High	Fixed
06	MasterHepa: unchecked math in getMultiplier(), withdraw()	High	Fixed
07	MasterHepa: reclaimTokenOwnership() transfers HEPA token ownership	High	Informed
08	MasterHepa: rewards updaters lacks safety guards	High	
09	Governance: unlock() always updates _lastUnlockBlock	Medium	Fixed
10	MasterHepa: userGlobalInfo sums different tokens	Medium	Informed
11	MasterHepa: updating parameters without safety guards	Medium	Fixed
12	Authorizable: no events	Low	Fixed
13	MasterHepa: no checks on input data	Low	Informed
14	MasterHepa: variables naming	Low	Informed
15	MasterHepa: variable declaration with 0 value	Low	Fixed
16	MasterHepa: checking boolean == value	Low	Fixed
17	MasterHepa: poolId1 variable duplicates poolExistence	Low	Fixed
18	MasterHepa: using struct for globalAmount	Low	Fixed
19	MasterHepa: no need in local variable in getGlobalAmount()	Low	Fixed
20	MasterHepa: excessive else condition in deposit()	Low	Fixed

21	MasterHepa: withdraw() code efficiency	Low	Informed
22	MasterHepa: firstDepositBlock, lastWithdrawBlock and blockDelta duplicate each other	Low	Fixed
23	MasterHepa: typo in starblockUpdate()	Low	Fixed
24	MasterHepa: blockDeltaStartStage and blockDeltaEndStage duplicate each other	Low	Informed
25	MasterHepa: getNewRewardPerBlock() needs a description	Low	Fixed
26	General recommendations	Low	

#01 Governance: no safety guards for lockFromBlock and lockToBlock High

Owner controls `lockFromBlock` and `lockToBlock` without any safeguards and can block unlocking user funds by setting unlock block to arbitrary big value with the function `lockFromUpdate()`.

Update: the issue was fixed by removing functions to update those values.

#02 Governance: canUnlockAmount() could return wrong value High

`canUnlockAmount()` function checks for `block.number < lockFromBlock` instead of `<_lastUnlockBlock[msg.sender]`. In such case, the function returns 0 for the amount that can be unlocked. We rate such issues as medium severity, but taking into consideration #01 the issue is classified as High.

Hepa team response: no need to fix. Misunderstanding here, `lockFromBlock` is a global variable equal to block when unlocking starts. See #09 for more details.

Commentary on the update: we were concerned about global variable under the owner's control. The way the #01 issue was fixed mitigates this issue as well.

#03 Governance: mint() is open for the owner High

`mint()` is open for the owner. The current realization of MasterHepa allows the transfer of token ownership. Any authorized by the MasterHepa's owner account can transfer ownership to another address and mint tokens to its address. It must be tokens that the maximum number of tokens to mint is capped by the `_cap` value in the token contract.

Hepa team response: no need to fix. Mint is needed for MasterHepa contract. To prevent infinite minting there is a CAP variable. It is enforced in `_beforeTokenTransfer` hook.

Commentary on the update: minting with MasterChef contracts is safe then ownership is given to them permanently. Current realisation of `reclaimTokenOwnership()` function of MasterHepa contract makes possible malicious minting by any of authorized addresses.

#04 MasterHepa: no safety guards for devFeeStage and userFeeStage High

The owner can withdraw all user's LP tokens to dev address. To do this they can set `devFeeStage` to an arbitrary big number, `userFeeStage` to zero, and withdraw all tokens from the pool. We recommend adding safety guards in functions that set these parameters.

Hepa team response: fixed by removing functions to update those values.

#05 MasterHepa: owner controls user.lastWithdrawBlock High

The owner or authorized address can change the user's `lastWithdrawBlock` and `firstDepositBlock` with revised functions to change the fee.

Update: the issue was fixed by removing functions to update those values.

#06 MasterHepa: unchecked math in getMultiplier(), withdraw() High

Unchecked math in `getMultiplier()` and `withdraw()` functions. Severity is High due to issues #05 and #10.

Update: the issue was fixed.

#07 MasterHepa: reclaimTokenOwnership() transfers HEPA token ownership High

`reclaimTokenOwnership()` transfers HEPA token to the new owner with the power of unlimited minting.

Hepa team response: no need to fix. This function is needed to regain token ownership in case of emergency or new MaterHepa contract deployment.

Commentary on the update: we recommend using additional authorization control for this function, i.e. Timelock or multisig.

#08 MasterHepa: rewards updaters lacks safety guards High

`rewardUpdate()` and `rewardMulUpdate()` functions are used for updating the crucial reward parameters. Although they could be called only by authorized addresses, we recommend adding safety guards — capping the new values. If the owner's account gets compromised or the owner acts maliciously, the attacker can set an arbitrary big value for the REWARD_PER_BLOCK variable. In such a case the amount of tokens till cap will be minted soon and token price will drop. It must be noted that the risks of this issue are the same as #03 and regards only the case when the owner's account is not trusted or is not properly secured.

#09 Governance: unlock() always updates _lastUnlockBlock Medium

`unlock()` function updates `_lastUnlockBlock[msg.sender]` even for zero amount. So unlocking percent could be positive then `block.number == lockFromBlock`. We can't understand the developer's intention: either the `lock()` function should write `_lastUnlockBlock[_holder] = block.number` or `unlock()` shouldn't do the same.

Hepa team response: fixed. `unlock()` function should update user `_lastUnlockBlock` value to the block when user unlocked, to calculate remaining unlock value inside `canUnlockAmount()` for the user. To address update on zero amount guard added.

`#10 MasterHepa: userGlobalInfo sums different tokens` Medium

`userGlobalInfo` should be a simple variable instead of struct. Moreover, it sums different lpTokens from different pools (possibly with different decimals).

Hepa team response: fixed. Only LP tokens should be allowed by the pools, and those has same decimals. Its owner responsibility to check it when adding the pool.

Commentary on the update: the fix is only the Hepa team being informed. No checks on the decimals are performed in `add()` function. Since `userGlobalInfo` has no other functions besides getting the total number of deposited tokens, the issue's severity should be considered as Low.

`#11 MasterHepa: updating parameters without safety guards` Medium

Updating parameters functions lack checks on input data (safety guards and arrays' lengths checking).

Update: the issue was fixed. Added safety guards.

`#12 Authorizable: no events` Low

Authorizable contract has no events nor getters for authorized addresses.

Update: the issue was fixed. Events added.

`#13 MasterHepa: no checks on input data` Low

No checks on input data in the constructor. Possible arrays' lengths mismatches.

Hepa team response: no need to fix.

`#14 MasterHepa: variables naming` Low

Confusing variable naming. UPPERCASE names should be used only for constants/immutables.

Hepa team response: no need to fix. Styles issue.

`#15 MasterHepa: variable declaration with 0 value` Low

No need to declare variables explicitly assigned to zero, see `totalAllocPoint`, `result`, `lockAmount`.

Update: the issue was fixed fixed.

#16 MasterHepa: checking boolean == value — Low

No need to check boolean variables equal true/false in require of if statements, see `nonDuplicated`.

Update: the issue was fixed.

#17 MasterHepa: poolId1 variable duplicates poolExistence — Low

`poolId1` variable duplicates the `poolExistence`. Only one of these should be used.

Update: the issue was fixed. Removed variable and references.

#18 MasterHepa: using struct for globalAmount — Low

`userGlobalInfo` should be uint variable instead of a structure.

Update: the issue was fixed.

#19 MasterHepa: no need in local variable in getGlobalAmount() — Low

`getGlobalAmount()` declares a local variable `current`, should simply return

`userGlobalInfo[_user].globalAmount`.

Update: the issue was fixed.

#20 MasterHepa: excessive else condition in deposit() — Low

`deposit()` function contains excessive if/else construction in L323, should be

```
if (user.firstDepositBlock == 0)
```

Update: the issue was fixed.

#21 MasterHepa: withdraw() code efficiency — Low

`withdraw()` function could be significantly refactored, e.g. L349-381 could be significantly reduced in size.

Hepa team response: no need to fix.

#22 MasterHepa: firstDepositBlock, lastWithdrawBlock and blockDelta duplicate each other Low

`user.blockDelta` is used only in `withdraw()` function and is not needed to be written. `user.lastWithdrawBlock` and `user.firstDepositBlock` duplicate each other and only one of these should be used.

Hepa team response: no need to fix.

#23 MasterHepa: typo in starblockUpdate() Low

Possibly should be named `startBlockUpdate()`.

Update: the issue was fixed.

#24 MasterHepa: blockDeltaStartStage and blockDeltaEndStage duplicate each other Low

Staging parameters are used only in `withdraw()` function and could be reduced to one array and combination of strict and non strict inequalities.

Hepa team response: no need to fix.

#25 MasterHepa: getNewRewardPerBlock() needs a description Low

`getNewRewardPerBlock()` function needs a description about using `pid1` parameter.

Update: the issue was fixed. Added description.

#26 General recommendations Low

Comments on MasterHepa.sol's L337, L341, L345, L349, L353, L357, L361, L365 are set for specific values of userFeeStage and devFeeStage variables. Although arbitrary values could be set in the constructor.

Variables result and .lockAmount should be initialized explicitly to zero in MasterHepa L191, L271.

Conclusion

7 high severity issues were found. We recommend fixing it before deployment to mainnet. The most important ones regard the ability of the owner being able to mint HepaToken and to withdraw user's staked LP tokens to the MasterHepa contract.

Audit includes recommendations on the code improving and preventing potential attacks.

Update: Hepa Finance team has responded to this report. Most of the issues were fixed. 3 high severity issues about minting remain in the updated code. It must be noted that these issues regard to the case when the owner of the contract is not trusted or it's account is not properly secured. Individual responses to the high severity issues were added after each item in the section.
Updated contracts are deployed to the testnet of BSC:
HepaToken `0x422322FBD72899A26eEF3e2b3e505a2e80E78fCB`,
MasterHepa `0x88Bacad4a016026863D9A96c424041998cdF09D9`.

Commentary on the update: another High severity issue #08 was found that was not included in the previous version of this document.

Appendix A. Issues' severity classification

We consider an issue to be critical, if it may cause unlimited losses, or breaks the workflow of the contract and could be easily triggered.

High severity issues may lead to limited losses or break interaction with users or other contracts under very specific conditions.

Medium severity issues do not cause the full loss of functionality but break the contract logic.

Low severity issues are typically nonoptimal code, unused variables, errors in messages. Usually, these issues do not need immediate reactions.

Appendix B. 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