HashEx
BLOCKCHAIN SECURITY

BRCStarter

smart contracts final audit report

January 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 BRCStarter team to perform an audit of their smart contract. The audit was conducted between 18/01/2024 and 23/01/2024.

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 is available in the @BRCStarter/Smart-contracts GitHub repository after the commit d5fe434.

2.1 Summary

Project name	BRCStarter
URL	https://brcstarter.io
Platform	Binance Smart Chain
Language	Solidity
Centralization level	● High
Centralization risk	● High

2.2 Contracts

Name	Address
BrcStarterTOKEN	
Farming	
Stake	
LaunchPad	
TokenDistributor	

3. Project centralization risks

The contracts are ownable, their governance functions can be maliciously or mistakenly used to make some public functions unusable.

C95CR0e Reward emission can be set to arbitrary rate

The contract owner can disable rewards or increase the emission rate to absurdly high value to drain reward's balance or to block them completely.

C96CR0f Owner's abuse of privileges

The contract owner assigns for the admin role, which bearers have the ability to explicitly increase user's staking weight by granting xpPoints (to reward Zealy quests and other campaigns, according to documentation). This can be abused to increase weight or to grant weight to non-staking addresses.

Owner can change address of the Farming contract, possibly breaking all user-interacting functions.

C97CR11 Owner's abuse of privileges

The owner or admins (assigned by owner) can set an arbitrary time period for refund delay and duration of the first round.

Admins are in charge of setting user's KYC status, but it can be reverted, i.e., mistakenly approving KYC user can't be prevented from participation in any future sales.

Admins can set refund timestamp in the past, making refunding impossible for the users.

C97CR10 Users donate their funds

The LaunchPad contract only collects users' ERC20 tokens in exchange for nothing. Users can only trust the owner if they are promised anything beyond that.

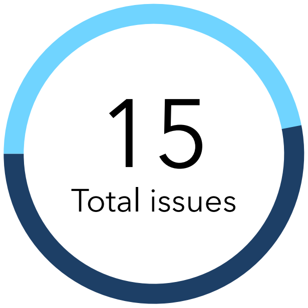
C98CR12 Owner's abuse of privileges

Contract held tokens may be locked if ownership is lost or compromised.

Contract admins have full access to the contract's balance of ERC20 tokens.

Total distributed balance, assigned by admins, may be greater than actual balance of the contract.

4. Found issues



●	Low	7 (47%)
●	Info	8 (53%)

C95. Farming

ID	Severity	Title	Status
C95I90	● Low	Gas optimizations	? Open
C95I91	● Info	Default visibility of state variables	? Open
C95I9d	● Info	Pool is not updated when the emission rate is changed	? Open

C96. Stake

ID	Severity	Title	Status
C96I93	● Low	Lack of input validation	? Open
C96I92	● Low	Gas optimizations	? Open
C96I95	● Info	Typographical error	? Open

C97. LaunchPad

ID	Severity	Title	Status
C97l97	Low	Gas optimizations	Open
C97l99	Low	Allocations for first round can be re-used	Open
C97l9a	Low	Incorrect return value	Open
C97l96	Info	Typographical error	Open
C97l98	Info	Two sale rounds intersect	Open

C98. TokenDistributor

ID	Severity	Title	Status
C98l9b	Low	Gas optimizations	Open
C98l9c	Info	Typographical error	Open
C98l9e	Info	Possible signature replay	Open
C98l9f	Info	Insufficient documentation for claimable amount calculation	Open

5. Contracts

C94. BrcStarterTOKEN

Overview

An ERC-20 standard token made with OpenZeppelin's implementation. Total supply of 21M tokens with 18 decimals is minted during the contract deployment to the owner's address.

C95. Farming

Overview

An extension contract for the Stake contract, Farming allow users of Stake to receive additional rewards meant to be paid in the same token as they're staking. Farming contract has no public functions and can be interacted only via Stake contract.

Issues

C95I90 Gas optimizations ● Low ⓘ Open

1. The `stakeContract` and `TOKEN` variables should be declared as immutables to save gas on reading them.

2. Multiple reads from storage in the `pendingRewards` function: `stakingPool.lastRewardDate`, `stakingPool.accTOKENPerShare` variables.

3. Multiple reads from storage in the `updatePool` function: `stakingPool.lastRewardDate` variable.

4. Multiple reads from storage in the `deposit` function: `user.amount`, `stakingPool.accTOKENPerShare` variables.

5. Multiple reads from storage in the `withdraw` function: `user.amount`, `stakingPool.accTOKENPerShare` variables.

6. Multiple reads from storage in the `claim` function: `user.amount`, `user.pendingRewards variables`.

C95I91 Default visibility of state variables ● Info ⓘ Open

The `farmingStartingDate` state variable in the contract has been declared without an explicit visibility specifier. In Solidity, if no visibility is specified, the default is `internal`. This means that while these variables are not directly accessible from external calls, they can be accessed and potentially modified by derived contracts. Not specifying visibility explicitly can lead to confusion about the intended accessibility of these variables and may inadvertently expose them to unintended modifications in future contract iterations.

C95I9d Pool is not updated when the emission rate is changed ● Info ⓘ Open

The contract has function to change emission rate.

```
    function changeEmission(uint256 _tokenPersec) external onlyOwner{
        tokenPerSecond = _tokenPersec;
        emit EmissionUpdated(_tokenPersec);
    }
```

This function does not call `updatePool` before changing the emission rate. This leads to a situation where rewards from the last pool update will be calculated with the new rate instead of the previous value.

C96. Stake

Overview

Simple staking contract for a single ERC20 token. First stake must satisfy the minimum amount requirement, which can be adjusted by the contract owner, subsequent stakes may be performed with an arbitrary amount. Deposit may have one of pre-defined locking periods to increase its effective weight obtained by the `getUserStakingData` function. Each consecutive user's deposit must not decrease locking period and also resets it. Locking period of the user can be increased explicitly without additional deposits. Participation in Stake contract allows users to receive rewards from the Farming contract, which can be received or compounded.

Contract has public getter `getUserStakingData` calculating the user's staking tier and the next level-up by the following rules:

available levels are 1 to 100,

arbitrary address has 1 level by default,

initial level gap is MINIMUM_STAKE_AMOUNT,

next level = current level + level gap,

level gap increases by MINIMUM_STAKE_AMOUNT for each 10 levels.

Issues

C96I93 Lack of input validation ● Low ⓘ Open

The `_stakingLockIndex` parameter of the `unStake`, `cancelCoolDown`, `extendLockPeriod` functions is not validated against the `STAKING_LOCK` array's length. Using a wrong parameter may result in transaction revert without a specific error message.

```
function unStake(uint256 _amount, uint256 _stakingLockIndex) external  {
  ...
  _userDatas[_sender].unlockDate = uint32(block.timestamp +
STAKING_LOCK[_stakingLockIndex])
  ...
}

function cancelCoolDown(uint256 _stakingLockIndex) external {
  ...
  _userDatas[_sender].unlockDate = uint32(block.timestamp +
STAKING_LOCK[_stakingLockIndex]);
  ...
}

function extendLockPeriod(uint256 _stakingLockIndex)external {
  ...
  _userDatas[_sender].unlockDate = uint32(block.timestamp +
STAKING_LOCK[_stakingLockIndex]);
}
```

C96I92 Gas optimizations ● Low ⓘ Open

1. Duplicated code: the `_isStaker` mapping duplicates the `EnumerableSet.AddressSet _stakers` functionality.

2. Unnecessary reads from storage in the `canUserUnstake` function: user's `UserData` structure is read in full but only 2 fields are used.

3. Multiple reads from storage in the stake function: `_userDatas[_sender].staked` variable.

4. Multiple reads from storage in the `unStake` function: `_userDatas[_sender].staked` variable.

5. Multiple reads from storage in the `_updateFarmingValue` function: `_userDatas[_sender].staked` variable.

C96I95 Typographical error ● Info ⓘ Open

The terms "Staking BRCST allow staker", "3 differents period", "additionnal", "wants to unstaking", "farmingreward", "avantages", "inited" are used in the contract, which is presumably a typographical error. The correct terms should be "Staking BRCST allows staker", "3 different periods", "additional", "wants to unstake", "farming reward", "advantages", "initialized". Misnaming variables can lead to confusion for developers, maintainers, and auditors, potentially obscuring the intent and functionality of the code.

C97. LaunchPad

Overview

Simple 2-round sale, payments in form of fixed ERC20 token can be made according to the allowances based on the deposits in the Stake contract. The only outcome for the user is the `getUserInvestForPool` getter for the user's invested amount.

Issues

C97I97 Gas optimizations ● Low ⓘ Open

1. Multiple reads from storage in the `createPool` function: `currentPoolId` variable.

2. Multiple reads from storage in the `registerToPool` function: `_poolDatas[_poolId].stakeWeightForPool` variable.

3. Multiple reads from storage in the `investInPoolRound1` function: `_userDatasForPool[_sender][_poolId].userAllocRound1`, `_poolDatas[_poolId].amountRaised` variables.

4. Multiple reads from storage in the `withdrawPoolFund` function: `_poolDatas[_poolId].refundOptionEnd` variable.

C97I99 Allocations for first round can be re-used

● Low ⓘ Open

The first sale round is limited by the user's allocations calculated with the `STAKINGCONTRACT.getStakerDatasForRegister(_user)` data. To obtain allocation one should stake tokens in the Stake contract, then register to the selected pool (sale) with the `registerToPool` function. Then staked funds can be withdrawn with delay `COOLDOWN_PERIOD` of 10 days, transferred to another user and re-used for obtaining allocation.

```
contract LaunchPad {
  function registerToPool(uint256 _poolId) external {
    ...
    (uint256 _stakeAmount,uint256 _level, uint256 _lockMultiplicator) =
        STAKINGCONTRACT.getStakerDatasForRegister(_user);
    require(_stakeAmount != 0,"You are not a staker");
    uint256 _userWeight =
        (_stakeAmount * _lockMultiplicator * (1000 + _level * MULT_TO_ADD))/1000;
    _poolDatas[_poolId].allocPerTokenStaked =
        (_poolDatas[_poolId].amountTarget * PRECISION) /
_poolDatas[_poolId].stakeWeightForPool;
    ...
  }
}

contract Stake {
  uint256[4] public STAKING_LOCK = [0, 91 days, 182 days, 365 days];

  function getStakerDatasForRegister(address _user) external view returns(uint256
stakeAmount,uint256 level, uint256 lockMultiplicator){
    stakeAmount = _userDatas[_user].staked;
    (level,) = _getLevel(stakeAmount + _userDatas[_user].userXp);
    lockMultiplicator =
      block.timestamp >= _userDatas[_user].unlockDate ?
        1 : STAKING_LOCK_MULTIPLICATOR[_userDatas[_user].stakingLockIndex];
  }

  function stake(uint256 _amount, uint256 _lockedPeriodIndex) external {
    ...
    require(TOKEN.transferFrom(_sender, address(this), _amount),"Transfer failed");
    uint256 _newLockCalculated = block.timestamp + STAKING_LOCK[_lockedPeriodIndex];
    _userDatas[_sender].staked = _amount;
    _userDatas[_sender].stakingLockIndex = uint8(_lockedPeriodIndex);
    _userDatas[_sender].unlockDate = uint32(_newLockCalculated);
    ...
  }

  function unStake(uint256 _amount, uint256 _stakingLockIndex) external {
    ...
    require(u.coolDownInited, "Need to init cool down first");
    ...
    require(TOKEN.transfer(_sender, _amount),"Transfer error");
}
```

C97I9a Incorrect return value ● Low Open

The `getUserAllocForPool` function sometimes returns user's calculated allocation regardless his KYC status. At the same time, user can't participate in pool sale without being approved by admin.

```
function getUserAllocForPool(address _user, uint256 _poolId) external view returns(uint256)
{
  ...
  if(block.timestamp >= p.startingDate + firstRoundDuration){
    (uint256 _stakeAmount,,) = STAKINGCONTRACT.getStakerDatasForRegister(_user);
    if(_stakeAmount != 0 && _hasKYC[_user]){
      return p.amountTarget - p.amountRaised;
    }else {
      return 0;
    }
  }else{
    if(u.userWeight == 0){
      return 0;
    }else if(u.userAllocRound1 != 0){
      return u.userAllocRound1;
    }else if(u.userInvest != 0){
      return u.userAllocRound1;
    }else{
      return _calculateUserAlloc(u.userWeight,p.allocPerTokenStaked);
    }
}
```

```
function registerToPool(uint256 _poolId) external {
  require(_hasKYC[_user],"Need to KYC to participate");
  ...
}
```

C97196 Typographical error

● Info ? Open

The terms "manualy", "allocs", "individualy", "garanty", "conbtract", "initing", "begining" are used in the contract, which is presumably a typographical error. The correct terms should be "manually", "allocations", "individually", "guarantee", "contract", "initializing", "beginning". Misnaming variables can lead to confusion for developers, maintainers, and auditors, potentially obscuring the intent and functionality of the code.

C97198 Two sale rounds intersect

● Info ? Open

Two sale rounds intersect at `block.timestamp == p.startingDate + firstRoundDuration`.

```
function investInPoolRound1(uint256 _poolId, uint256 _amount) external {
  ...
  require(_now <= p.startingDate + firstRoundDuration, "Round1 is finished");
  ...
}

function investInPoolFcfs(uint256 _poolId, uint256 _amount) external {
  ...
  require(block.timestamp >=p.startingDate+firstRoundDuration,"Fcfs didn't start");
  ...
}
```

C98. TokenDistributor

Overview

A vesting contract with individual schedules for different pool from the LaunchPad contract. TokenDistributor is designed to be operated with external authorization model.

Issues

C98I9b Gas optimizations ● Low ⓘ Open

1. Multiple reads from storage in the `setTokenDistribution` function: `_tokenDistribution[_fromPoolId].tge` variable.

2. Multiple reads from storage in the `_validateTransfer` function: `_userVesting[_sender][_poolId].totalClaimed` variable.

C98I9c Typographical error ● Info ⓘ Open

The terms "Lengths doesn't match", "valide" are used in the contract, which is presumably a typographical error. The correct terms should be "Lengths don't match", "valid". Misnaming variables can lead to confusion for developers, maintainers, and auditors, potentially obscuring the intent and functionality of the code.

C98I9e Possible signature replay ● Info ⓘ Open

The signature that checks the contract does not contain chain id.

```
function _checkMessage(
    address _user,
    uint256 _poolId,
    bytes32 _message
     )
```

```
        private pure returns (bool){
        return(keccak256(abi.encodePacked(_user,' renounced refund for poolId:
',_poolId))==_message);
    }
```

If there are several Launchpad contracts in different networks the signature created for one of them may be used for others. Consider adding Launchpad contract address and network id to the message to sign.

C98I9f Insufficient documentation for claimable amount calculation ● Info ⓘ Open

The function `getActualClaimable`, which calculates the amount a user can claim, is missing documentation. This is particularly important regarding the calculation of the claimable amount on subsequent calls. Currently, the amount for subsequent claims is calculated from the time of the last claim, rather than from the end of the previous cadence period. This approach may not align with the originally intended coding logic.

```
function _getActualClaimable(TokenDistributionModel memory _distrib,VestingModel memory
_vesting, uint256 _now)
  internal pure returns(uint256 claimable){
    if(_distrib.tge == 0 || _distrib.tge > _now || !_vesting.initiated){
      ...
    }else if(_vesting.lastClaimDate == 0){
      ...
    }else {
      uint256 _timePassed = _now - _vesting.lastClaimDate;
      if(_timePassed < _distrib.cliffPeriod){
        claimable = 0;
      }else{
      ...
    }
}
```

6. Conclusion

7 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.

The audited repository has tests only for the Stake contract. Some tests for Stake contract fail. We strongly recommend rigorously testing all the contract with unit and functional tests.

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