#HashEx
BLOCKCHAIN SECURITY

Habibi Finance

smart contracts preliminary audit report for internal use only

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 Habibi Finance team to perform an audit of their smart contract. The audit was conducted between 02/01/2024 and 04/01/2024.

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 scope of the audit includes two smart contracts: Stake and LaunchPad. SHA-1 hashes of the audited files are:Stake.sol `4c5c83393f05cba8747210ed98f9e2e5925a7922`LaunchPad.sol `7e71c7659df4c8c191a3af5f2e3a9bfd3b02f3c7`

Update. The Habibi Finance team has responded to this report. The updated code is located in the https://github.com/Habibi-Finance/habibi-launchpad repository and was checked after the commit 00c7964.

2.1 Summary

Project name	Habibi Finance
URL	https://habb.finance
Platform	Binance Smart Chain
Language	Solidity
Centralization level	High
Centralization risk	High

2.2 Contracts

Name	Address
Stake	
LaunchPad	

3. Project centralization risks

C87CR0b The owner can change launchpad contract address

The contract has a function `snapShotPool()` which can be called only by the launchpad address. The owner of the contract can change the launchpad address. Updating it may make the existing launchpad being unable to make pool snapshots.

C88CR0c The owner of the contract can withdraw deposited funds before the pool is finished

1. The contract has an emergencyWithdraw() function which allows the contract owner to withdraw any ERC20 token from the contract anytime.

```
function emergencyWithdraw(IERC20 _token, uint256 _amount) external onlyOwner {
    _token.transfer(
        _msgSender(),
        _amount != 0 ? _amount : _token.balanceOf(address(this))
    );
}
```

2. The owner can change round duration.

3. The owner can change the allocation amount for the second round.

4. The owner can change the tiers allocation.

5. The owner can stop pool deposits (finish the pool).

4. Found issues

High	1 (7%)
Medium	2 (13%)
Low	9 (60%)
Info	3 (20%)

C87. Stake

ID	Severity	Title	Status
C87I76	Low	Default visibility of state variables	Resolved
C87I78	Low	Lack of message in a require statement	Resolved
C87I75	Low	Gas optimizations	Partially fixed
C87I7d	Low	The result of token transfer is not checked	Resolved
C87I77	Info	Typographical error	Resolved
C87I79	Info	Commented out code	Resolved
C87I7f	Info	Inconsistent comment	Resolved

C88. LaunchPad

ID	Severity	Title	Status
C88I80	High	Setting tiers allocation makes impossible to create further snapshots	Resolved
C88I82	Medium	Wrong require check condition in the withdrawPoolFound() function	Resolved
C88I7a	Medium	Pool snapshot can be called twice for the same pool	Resolved
C88I7b	Low	Typographical errors	Resolved
C88I83	Low	Funds can be withdrawn by the owner regardless pool's state	Resolved
C88I7c	Low	Lack of message in a require statement	Resolved
C88I7e	Low	Lack of event	Resolved
C88I81	Low	Gas optimizations	Acknowledged

5. Contracts

C87. Stake

Overview

This smart contract is a staking contract, designed for users to stake a specific ERC20 token. The contract includes functionality for users to stake tokens, unstake tokens after a lock period, and manage multiple tiers of staking with different minimum amounts.

The first user's stake should be at least eligible for tier 1 amount, e.g. have an amount bigger than 10 million tokens. After the user makes a stake, he can withdraw only after a lock period.

Issues

C87I76 Default visibility of state variables

Low Resolved

Several state variables have been declared without explicit visibility. In Solidity, the default visibility for state variables is `internal`. However, relying on default visibility can lead to misunderstandings and potential security vulnerabilities if not carefully considered and documented.

- `_stakers` (of type `EnumerableSet.AddressSet`)
- `_isStaker` (mapping from address to bool)
- `_snapShotNumber` (uint256)
- `_tiersStaked` (uint256 array)

C87I78 Lack of message in a require statement

Low Resolved

A `require` statement in the `stake` function is missing descriptive an error message, which are crucial for debugging and understanding the reason for transaction reversion. Adding clear and concise messages to these statements will enhance error tracking and improve the overall

developer and user experience.

```
function stake(uint256 _amount) external {
    address _sender = _msgSender();
    require(TOKEN.transferFrom(_sender, address(this), _amount));
    ...
}
```

C87I75 Gas optimizations

Low Partially fixed

- the variable `lockPeriod` can be declared immutable. This will save gas on reading the variable.

- Excessive reads from storage in in the `getUserData()`, `canUserUnstake()`, `stake()`, `_snaps()` functions: only `_userSnapshots[user].length` and `_userSnapshots[user][length-1]` are needed but full array `_userSnapshots[user]` is read

C87I7d The result of token transfer is not checked

Low Resolved

The function unstake() transfers tokens but does not check the result of this tt

```
function unStake(uint256 _amount) external returns (bool) {
    ...
    return (TOKEN.transfer(_sender, _amount));
}
```

We recommend adding a require statement to check that the transfer function returned true.

C87I77 Typographical error

Info Resolved

The smart contract code contains a typographical error in the event name `LaunchpadSetted`, which should be correctly spelled as `LaunchpadSet` to align with standard English grammar conventions.

C87I79 Commented out code

Info Resolved

The smart contract contains commented-out code, specifically within the `unStake` function. This unused code can lead to confusion and clutter in the codebase. For cleanliness, maintainability, and clarity, it's recommended to remove any sections of code that are commented out and no longer in use.

```
function unStake(uint256 _amount) external returns (bool) {
    address _sender = _msgSender();
    uint256[] memory _snaps = _userSnapshots[_sender];
    //require(_snaps.length != 0, "You are not a staker");
    ...
 }
```

C87I7f Inconsistent comment

Info Resolved

The function `unstake()` has a comment for development purposes which is irrelevant for the production code.

```
function unStake(uint256 _amount) external returns (bool) {
    ...
    // do I have to reload staking period ?

    ...
}
```

C88. LaunchPad

Overview

The LaunchPad contract is designed as a fundraising platform where users can invest stablecoins in various pools. Each pool has a target amount to raise, allocation rules based on user stakes in a separate staking contract, and a snapshot mechanism to capture the state of stakes at a specific time.

There are two rounds to invest in a pool. In the first round user's allocation is calculated from his stake, in the second round the allocation is set to the same value for all users with any amount of allocation.

Issues

C88I80 Setting tiers allocation makes impossible to create further snapshots

High Resolved

There is a discrepancy in the size of `tiersAlloc[]` array. The array is initialized with 6 elements and the function `snapShotPool()` sets the elements from 2 to 6 (1 to 5 indexes).

```
function snapShotPool(
        uint256 _poolId
    ) external onlyOwner {
        (
            uint256[6] memory tiersStaked,
            uint256 snapShotNb
        ) = STAKINGCONTRACT.snapShotPool();

        PoolData storage pool = _poolDatas[_poolId];

        pool.snapShotNb = snapShotNb;

        uint256 allocForPool = _poolDatas[_poolId].amountTarget * PRECISION;
```

```
        pool.allocByTokenForTiers[1] = tiersStaked[1] != 0 ?
((allocForPool*tiersAllocs[1]/100) / tiersStaked[1]) : 0;
        pool.allocByTokenForTiers[2] = tiersStaked[2] != 0 ?
((allocForPool*tiersAllocs[2]/100) / tiersStaked[2]) : 0;
        pool.allocByTokenForTiers[3] = tiersStaked[3] != 0 ?
((allocForPool*tiersAllocs[3]/100) / tiersStaked[3]) : 0;
        pool.allocByTokenForTiers[4] = tiersStaked[4] != 0 ?
((allocForPool*tiersAllocs[4]/100) / tiersStaked[4]) : 0;
        pool.allocByTokenForTiers[5] = tiersStaked[5] != 0 ?
((allocForPool*tiersAllocs[5]/100) / tiersStaked[5]) : 0;


    }
```

However, if the contract owner calls `setTiersAllocs()`, the `tiersAlloc[]` array is reassigned to array of length 5. After that any call to `snapShopPool()` will fail due to trying to access the sixth element of the array.

```
    function setTiersAllocs(uint256[5] calldata _allocs) external onlyOwner {
        require(_allocs[0] + _allocs[1] + _allocs[2]+ _allocs[3]+ _allocs[4] == 100,
"Maths not good");
        emit TiersAllocSetted(_allocs);
        tiersAllocs = _allocs;
    }
```

Also it should be noted that the `setTiersAllocs()` sets the allocation for zero tier which is not used.

Recommendation

Set tiers allocations for indexed from 1 to 5 in the `setTiersAllocs()`. Or use the `tiersAllocs` with length of 5 everywhere in the contract.

C88I82 Wrong require check condition in the withdrawPoolFound() function

Medium Resolved

The `withdrawPoolFound(uint256 _poolId)` function is intended to allow the contract owner to

withdraw funds once a pool is finished. However, the function incorrectly checks if the pool is not finished (`require(!p.isFinished, "pool isn't finished yet")`).

```
    function withdrawPoolFound(uint256 _poolId) external onlyOwner {
        PoolData storage p = _poolDatas[_poolId];
        require(!p.foundWithdrawed, "founds already withdraw");
        require(!p.isFinished, "pool isn't finished yet");
        p.foundWithdrawed = true;

        STABLE.transfer(
            _msgSender(),
            p.amountRaised         );
    }
```

Recommendation

Change the check to `require(!p.isFinished, "pool isn't finished yet");`

C88I7a Pool snapshot can be called twice for the same pool

Medium Resolved

The function `snapShotPool(uint256 _poolId)` currently lacks restrictions to prevent it from being called multiple times for the same pool. If invoked more than once, there could arise discrepancies in allocation calculations.

If some users make investments before the second snapshot call, their allocation would be calculated based on the tiers config and stakes amount during the first snapshot.

Other users will use new tiers config and stake amounts.

This can lead to a situation when total user allocation won't sum up to the allocation for the pool. They may be bigger or less than the desired amount.

```
    function snapShotPool(
        uint256 _poolId
    ) external onlyOwner {
```

```
        (
            uint256[6] memory tiersStaked,
            uint256 snapShotNb
        ) = STAKINGCONTRACT.snapShotPool();

        PoolData storage pool = _poolDatas[_poolId];

        pool.snapShotNb = snapShotNb;
        ...
    }
```

Recommendation

C88I7b Typographical errors

Low Resolved

The terms "setted", "foundWithdrawed", "to much", "found", "already withdraw" are used in the contract, which is presumably a typographical error. The correct terms should be "set", "foundWithdrawn", "too much", "fund", and "already withdrawn". Misnaming variables can lead to confusion for developers, maintainers, and auditors, potentially obscuring the intent and functionality of the code.

C88I83 Funds can be withdrawn by the owner regardless pool's state

Low Resolved

The `emergencyWithdraw(IERC20 _token, uint256 _amount)` function allows the contract owner to withdraw any amount of the specified token from the contract at any time, without considering the state of any pool. While this provides a broad power for emergencies, it also bypasses all checks and balances put in place for the standard withdrawal process, including whether a pool is finished.

```
    function emergencyWithdraw(IERC20 _token, uint256 _amount) external onlyOwner {
        _token.transfer(
            _msgSender(),
            _amount != 0 ? _amount : _token.balanceOf(address(this))
        );
```

```
    }
```

C88I7c Lack of message in a require statement

Low Resolved

A `require` statement in the `investInPool` function is missing descriptive an error message, which are crucial for debugging and understanding the reason for transaction reversion. Adding clear and concise messages to these statements will enhance error tracking and improve the overall developer and user experience.

C88I7e Lack of event

Low Resolved

The function `closePool()` changes important state variable but does not emit an event.

```
    function closePool(uint256 _poolId) external onlyOwner {
        PoolData storage p = _poolDatas[_poolId];
        require(p.startingDate != 0, "Pool doesn't exist");
        p.isFinished = true;
}
```

C88I81 Gas optimizations

Low Acknowledged

Unnecessary read from storage in the `investInPool()` function: `_userInvest[_sender][_poolId]`, `_poolDatas[_poolId].amountRaised` variables

6. Conclusion

1 high, 2 medium, 9 low severity issues were found during the audit. 1 high, 2 medium, 7 low 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