#HashEx
BLOCKCHAIN SECURITY

Fringe Finance Staking

smart contracts
final audit report

August 2022

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 Fringe Finance team to perform an audit of their staking smart contracts. The audit was conducted between 01/08/2022 and 05/08/2022.

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 audited contracts are designed to be deployed with proxies. Users have no choice but to trust the owners, who can update the contracts at their will.

The code is available at the GitHub repository @fringe-finance/usb-staking-smart-contracts after the commit 81c437e.

Update: the Fringe Finance team has responded to this report. The updated code is located in the same repository after the 9b7a858 commit.

2.1 Summary

Project name	Fringe Finance Staking
URL	https://fringe.fi
Platform	Ethereum
Language	Solidity

2.2 Contracts

Name	Address
Multiple contracts	
USBStakingFactory	https://github.com/fringe-finance/usb-staking-smart-contracts/blob/81c437ec3a8c00e0a6d7b58f843f6ebf79d28f3e/contracts/USBStakingFactory.sol
USBStakingYield	https://github.com/fringe-finance/usb-staking-smart-contracts/blob/81c437ec3a8c00e0a6d7b58f843f6ebf79d28f3e/contracts/USBStakingYield.sol
USBStaking	https://github.com/fringe-finance/usb-staking-smart-contracts/blob/81c437ec3a8c00e0a6d7b58f843f6ebf79d28f3e/contracts/USBStaking.sol

3. Found issues

Critical	1 (5%)
High	1 (5%)
Medium	2 (11%)
Low	7 (37%)
Info	8 (42%)

C1. Multiple contracts

ID	Severity	Title	Status
C1-01	Medium	transferAdminship() allows renouncing adminship	Resolved
C1-02	Low	Gas optimization	Partially fixed
C1-03	Info	Floating Pragma	Partially fixed
C1-04	Info	Legacy OpenZeppelin contracts	Acknowledged

C2. USBStakingFactory

ID	Severity	Title	Status
C2-01	Low	Few events	Resolved
C2-02	Low	Library not used	Resolved

C3. USBStakingYield

ID	Severity	Title	Status
C3-01	Low	Few events	Resolved
C3-02	Info	Typos	Resolved
C3-03	Info	Admins can sweep reward tokens	Acknowledged
C3-04	Info	Gas optimization	Partially fixed

C4. USBStaking

ID	Severity	Title	Status
C4-01	Critical	claimRewardAndStake() interferes with yield rewards calculations	Resolved
C4-02	High	emergencyUnstake() execution can be suspended	Resolved
C4-03	Medium	emergencyUnstake() does not acknowledge rewards	Resolved
C4-04	Low	Gas optimization	Partially fixed
C4-05	Low	setParams() should call for internal set functions	Resolved
C4-06	Low	Few events	Acknowledged
C4-07	Info	Admins can sweep reward tokens	Acknowledged
C4-08	Info	Possible reentrancy	Acknowledged
C4-09	Info	Typos	Resolved

4. Contracts

C1. Multiple contracts

Overview

The following issues are related to multiple contracts.

Issues

C1-01 transferAdminship() allows renouncing adminship

Medium Resolved

When the admin's address is passed as a parameter to the `transferAdminship()` method of the USBStaking and USBStakingFactory contracts, the execution reaches the `_revokeRole()` for the admin, effectively renouncing the ownership.

```
function transferAdminship(address _admin) public onlyAdmin {
    require(_admin != address(0), "USBStakingFactory: _admin is zero");
    _grantRole(DEFAULT_ADMIN_ROLE, _admin);
    _revokeRole(DEFAULT_ADMIN_ROLE, msg.sender);
}
```

Recommendation

Check if the new admin address is different from the stored one.

C1-02 Gas optimization

Low Partially fixed

Most of the public functions could be declared external to save gas on calling them.

C1-03 Floating Pragma

Info Partially fixed

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

C1-04 Legacy OpenZeppelin contracts

Info Acknowledged

Update the OpenZeppelin contracts to the latest version and use the `_disableInitializers()` function in the proxy logic constructor to prevent reinitialization.

C2. USBStakingFactory

Overview

This contract deploys new USBStaking contracts through BeaconProxy. Staking admins can add new USBStakingYield contracts to the USBStaking under their governance.

Issues

C2-01 Few events

Low Resolved

Many functions from the contract lack corresponding events:

1. `deployStaking()`
2. `addYieldToStaking()`
3. `removeYieldStaking()`

C2-02 Library not used

Low Resolved

The functionality of the `PausableUpgradeable` library is not used in this contract.

C3. USBStakingYield

Overview

Contract for additional optional rewards for USBStaking users. Governed by the admins of the parent USBStaking contract. Sources of rewards are unclear, rewards can be canceled by the admins though the pending reward would be calculated anyway.

Issues

C3-01 Few events

Low Resolved

Many functions from the contract lack events:

1. `_setYieldRewardPerBlock()`
2. `_setYieldPeriod()`

C3-02 Typos

Info Resolved

Typos reduce the code's readability:

1. `L135, L137` 'transfered' should be replaced with 'transferred'
2. `L102` No code entity named 'USBStakingFixed' was found, should be replaced with 'USBStaking'.

C3-03 Admins can sweep reward tokens

Info Acknowledged

Admins of USBStakingYield can sweep any reward tokens at any time. This may affect the abrupt stop of the issuance of reward tokens to users.

C3-04 Gas optimization

Info Partially fixed

1. Multiple reads storage variables in:- modifier `onlyAdmin()`: `usbStaking` variable- modifier `onlyManager()`: `usbStaking` variable- function `_updatePendingYieldReward()`: `userYield.instantAccumulatedShareOfYieldReward` variable- function `getUnclaimedRewardAmount()`: `totalClaimedYieldReward` variable- function `getBlockDelta()`: `startBlock`, `endBlock` variables- function `getUserYieldPosition()`: `usbStaking`, `lastYieldRewardBlock` variablesYou can save some gas by creating a separate variable for some of the fields, or by accessing the `memory` copy of the variables.
2. The functionality of the `AccessControlUpgradeable.sol` and `USBStakingFactory` libraries is not used in this contract. If remove these imports, then can save some amount of gas when deploying the contract.
3. Possible consecutive writes in `claimYieldReward()` and `_safeYieldRewardTokenTransfer()` : `totalClaimedYieldReward`, `userYield.claimedYieldReward`, and `userYield.pendingYieldReward` are rewritten in case of shortfall.

C4. USBStaking

Overview

This contract has the functionality of staking specified coins. Primary rewards are accrued within the USBStaking contract, optional rewards could be added through the array of USBStakingYield contracts (up to 256 in length). Sources of rewards are unclear, rewards can be canceled by the admins though the pending reward would be calculated anyway.

Issues

C4-01 claimRewardAndStake() interferes with yield rewards calculations

Critical Resolved

The public available `claimRewardAndStake()` function allows the user to compound their rewards if the reward token is the same as the staking token. However, this function updates the `user.stake` amount, adding the compounded value.

```
    function claimRewardAndStake() public whenNotPaused {
        require(address(rewardToken) == address(stakeToken), "USBStaking: not allowed for
this product");
        update();
        UserPosition storage user = userPosition[msg.sender];
        uint256 accumulatedShareOfReward = getAccumulatedShareOfReward(msg.sender);
        if (accumulatedShareOfReward > user.instantAcumulatedShareOfReward) {
            user.pendingReward += accumulatedShareOfReward -
user.instantAcumulatedShareOfReward;
        }
        uint256 pendingReward = user.pendingReward;
        require(pendingReward > 0, "USBStaking: pendingReward=0");
        require(pendingReward <= getRewardTokenAmount(), "USBStaking: insufficient amount
of rewardToken");
        totalClaimedReward += pendingReward;
        user.claimedReward += pendingReward;
        totalStake += pendingReward;
        user.stake += pendingReward;
        user.pendingReward = 0;
        user.instantAcumulatedShareOfReward = getAccumulatedShareOfReward(msg.sender);
        emit ClaimRewardAndStake(msg.sender, pendingReward);
    }
```

But at the same time, the `claimYieldReward()` function of the USBStakingYield contract accrues the user's yield reward by reading this freshly updated value.

```
    function claimYieldReward() public whenNotPaused {
        update();
        _updatePendingYieldReward(msg.sender);
        _updateInstantAccumulatedShareOfYieldReward(msg.sender);
        ...
    }

    function _updatePendingYieldReward(address user) internal {
        ...
        uint256 accumulatedShareOfYieldReward = getAccumulatedShareOfYieldReward(user);
        ...
    }

    function getAccumulatedShareOfYieldReward(address user) public view returns (uint256)
{
```

```
        uint256 accumulatorMultiplier = getAccumulatorMultiplier();
        (uint256 userStake,,,) = usbStaking.userPosition(user);
        return userStake * accumulatedYieldRewardTokenPerStakeToken /
accumulatorMultiplier;
    }
```

The resulting yield reward amount for a single user can exceed the total amount of yield rewards `(endBlock-startBlock)*yieldRewardPerBlock`.

Recommendation

Update user's pending yield rewards and instant accumulated shares of yield rewards during the `claimRewardAndStake()` execution.

C4-02 emergencyUnstake() execution can be suspended ● High ✓ Resolved

The `update()` execution in `emergencyUnstake()` is obligatory when the contract is not paused. When the update logic goes wrong in either of the USBSTakingYield contracts, `safeTransfer()` may never be reached.

```
    function emergencyUnstake() public {
        if (!paused()) {
            update();
        }
        ...
        stakeToken.safeTransfer(address(msg.sender), userStake);
    }

    function update() public whenNotPaused {
        ...
        for(uint8 i = 0; i < yieldsLength;) {
            yields[i].update();
            unchecked { ++i; }
        }
    }
```

Malicious or hacked admin may add failing yield address and leave the USBSTaking contract

unpaused, denying users from the emergency exit.

Recommendation

Consider wrapping the `update()` call into `try/catch` statements or removing it completely by adding an update flag to the function's parameters, or even simply leave the decision of calling `update()` prior to `emergencyUnstake()` to the user.

C4-03 emergencyUnstake() does not acknowledge rewards

Medium Resolved

When the `emergencyUnstake()` function is called, `pendingReward` is set to zero with no respect to the users, canceling their claimable rewards.

```
function emergencyUnstake() public {
    ...
    user.pendingReward = 0;
    ...
}
```

Leaving the pending reward amount untouched would allow the user to receive their accrued rewards later.

Also, `getUnclaimedRewardAmount()` function doesn't account rewards of emergency unstaked users.

Recommendation

Consider leaving user's pending reward unmodified.

C4-04 Gas optimization

Low Partially fixed

1. Multiple reads storage variables in:- function `sweepTokens()`: `stakeToken`, `totalStake` variables- function `removeYield()`: `yields.length` variable- function `update()`: `totalStake` variable- function `stakeTo()`: `user.instantAcumulatedShareOfReward` variable - function `unstake()`: `user.stake`, `user.instantAcumulatedShareOfReward` variables - function `claimReward()`: `user.instantAcumulatedShareOfReward` variable- function `claimRewardAndStake()`: `user.instantAcumulatedShareOfReward`, `user.pendingReward` variables- function `exit()`: `user.instantAcumulatedShareOfReward` variable- function `getRewardTokenAmount()`: `totalStake` variable- function `getBlockDelta()`: `startBlock`, `endBlock` variables- function `getUserPosition()`: `lastRewardBlock`, `totalStake`, `user.instantAcumulatedShareOfReward`, `user.stake` variablesYou can save some gas by creating a separate variable for some of the fields, or by accessing the `memory` copy of the variables.
2. Possible consecutive writes in `claimReward()` and `_safeRewardTokenTrasfer()`: `totalClaimedReward`, `user.claimedReward`, and `user.pendingReward` are rewritten in case of shortfall.

C4-05 setParams() should call for internal set functions

Low Resolved

The `setParams()` function changes `rewardPerBlock`, `startBlock`, `endBlock` by calling the `setRewardPerBlock()` and `setPeriod()` public functions. But these functions in turn call the internal functions `_setRewardPerBlock()` and `_setPeriod()`. It will be more economical on gas to immediately call internal functions.

```
    function setRewardPerBlock(uint256 _rewardPerBlock) public onlyManager {
        _setRewardPerBlock(_rewardPerBlock);
    }

    function setPeriod(uint256 _startBlock, uint256 _endBlock) public onlyManager {
        _setPeriod(_startBlock, _endBlock);
    }

    function setParams(uint256 _rewardPerBlock, uint256 _startBlock, uint256 _endBlock)
public onlyManager {
        setRewardPerBlock(_rewardPerBlock);
        setPeriod(_startBlock, _endBlock);
    }
```

C4-06 Few events

Low Acknowledged

Several functions from the contract lack events:

1. `_setRewardPerBlock()`
2. `_setPeriod()`
3. `setStakePaused()`

C4-07 Admins can sweep reward tokens

Info Acknowledged

Admins of USBStaking can sweep any reward tokens at any time. This may affect the abrupt stop of the issuance of reward tokens to users.

C4-08 Possible reentrancy

Info Acknowledged

Hypothetically reward tokens may contain the implementation of a callback on tokens sent or received. In this case, the reentrancy on token transfers is possible in `stakeTo()` and `unstake()` functions, allowing the user to duplicate the latest pending yield reward amount by double calling the `updatePendingYieldReward()` function of USBStakingYield contract.

Admins of USBStakingFactory must avoid adding staking contracts for tokens with reentrancy hooks.

C4-09 Typos

Info Resolved

Typos reduce the code's readability:

1. `L22` 'by' should be replaced with 'be'
2. `L208, L210` 'transfered' should be replaced with 'transferred'
3. `L250-L255` '_yiedlId' should be replaced with '_yieldId'
4. `L437, L499, L529` '_safeRewardTokenTrasfer' should be replaced with '_safeRewardTokenTransfer'

5. Conclusion

1 critical, 1 high, 2 medium, 7 low severity issues were found during the audit. 1 critical, 1 high, 2 medium, 4 low issues were resolved in the update.

The reviewed contract is highly dependent on the owner's account. Users using the project have to trust the owner and that the owner's account is properly secured.

The audited contracts are designed to be deployed with proxies. Users have no choice but to trust the owners, who can update the contracts at their will.

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. 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

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY