#HashEx
BLOCKCHAIN SECURITY

Frensly

smart contracts final audit report

November 2023

hashex.org

contact@hashex.org

Contents

1. Disclaimer 3

2. Overview 4

3. Project centralization risks 6

4. Found issues 7

5. Contracts 9

6. Conclusion 17

Appendix A. Issues' severity classification 18

Appendix B. Issue status description 19

Appendix C. List of examined issue types 20

Appendix D. Centralization risks classification 21

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 Frensly team to perform an audit of their smart contract. The audit was conducted between 24/10/2023 and 26/10/2023.

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 file without tests of any type. The SHA-1 hashes of audited contracts:

frensly.sol `847c7f4a47cf64073a432d95840c3e54baa4778b`.

Update: the Frensly team has responded to this report. The updated contract has SHA-1 of `35aa4f7d2d7f254b39a53b5c0d0a8ca0e5e67949`. Updated code is deployed in the Base chain at 0x66fA4044757Fb7812EF5b8149649d45d607624E0.

2.1 Summary

Project name	Frensly
URL	https://frensly.io
Platform	Base
Language	Solidity
Centralization level	High
Centralization risk	High

2.2 Contracts

Name	Address
Frensly	0x66fA4044757Fb7812EF5b8149649d45d607624E0

3. Project centralization risks

The project owner can set a malicious fee destination and significantly increase gas spends for deposits and/or withdrawals, in general or selectively.

Anyone can create their own pool to receive part of fees and, therefore, to control amount of gas spent for its deposit/withdraw activities. Users must check the pool creator address before using that pool.

4. Found issues

High	1 (9%)
Low	6 (55%)
Info	4 (36%)

C3f. Frensly

ID	Severity	Title	Status
C3flb8	High	Possible DoS attack of subject and protocolFeeDestination accounts	Partially fixed
C3flb9	Low	Wrong price value in Trade event	Resolved
C3flb2	Low	Usage of default visibility for state variables	Resolved
C3flb4	Low	Non-indexed parameters in events	Resolved
C3flb6	Low	Lack of events	Resolved
C3flb7	Low	Possible wrong fee recipient	Resolved
C3flb1	Low	Gas optimizations	Resolved
C3flb5	Info	Typographical errors	Resolved
C3flb3	Info	Unconventional naming for a constant variable	Resolved

C3flba	Info	Unaddressed TODO comments	Resolved
C3flbb	Info	Lack of NatSpec documentation	Resolved

5. Contracts

C3f. Frensly

Overview

A referral staking contract for native EVM coin with fixed fees of 2.5% to protocol, 2.5% to eligible users, and 5% to creator of the pool.

Issues

C3f1b8 Possible DoS attack of subject and protocolFeeDestination accounts

High Partially fixed

The `_distributeFees` function, invoked within the buy and sell functions, transfers native currency to the `protocolFeeDestination` and `sharesSubject` addresses. If either of these addresses is a malicious contract, it can be coded to revert upon receiving funds, potentially causing the entire buy or sell transaction to fail. An attacker can exploit this to disrupt the contract's operations selectively.

```
function _distributeFees(
        address sharesSubject,
        uint256 subjectFee,
        uint256 protocolFee,
        uint256 holderFee
    ) internal {
        (bool success1, ) = protocolFeeDestination.call{value: protocolFee}("");
        (bool success2, ) = sharesSubject.call{value: subjectFee}("");
        require(success1 && success2, "Unable to send funds");

        if (holderFee > 0) {
            _distributeFeeToHolders(sharesSubject, holderFee);
        }
    }
```

Recommendation

Restrict setting protocol fee destination and shares subjects addresses to be non-contracts or give up the strict requirement of call success.

Update

Removing strict requirement of call success must be followed by using a fixed gas value for external call.

C3f1b9 Wrong price value in Trade event

Low Resolved

In the `buyShares()` function, the price, as well as associated fees, is initially calculated basing on the current supply. However, for the `Trade` event, the price is re-calculated using an updated supply (`supply + amount`). This inconsistency can lead to discrepancies in the recorded price in the event log compared to the actual transactional value, potentially causing confusion and misrepresentation of actual transaction metrics.

```
    function buyShares(address sharesSubject, uint256 amount) nonReentrant public payable
{
        ...
        (
            uint256 price,
            uint256 protocolFee,
            uint256 subjectFee,
            uint256 holderFee
        ) = _getFees(sharesSubject, supply, amount);
        ...
        //emits the event beforehand
        emit Trade(
            msg.sender,
            sharesSubject,
            true,
            amount,
            price,
            protocolFee,
            subjectFee,
            holderFee,
```

```
            supply + amount,
            getPrice(supply + amount, 1e6) //@audit wrong second param?
        );
        ...
    }
```

Update

The team responded that the price in event is used to track the new price after purchase.

C3fIb2 Usage of default visibility for state variables

Low Resolved

The smart contract contains several state variables (`decMul`, `subjectHolderToNextIndex`, `subjectHolderToPrevIndex`, `subjectToIndexToHolder`, `subjectToMaxIndex`, `subjectToLatestValidIndex`) that use the default visibility. Default visibility for state variables is considered `internal`. While `internal` variables can't be read from external contracts or web3 calls, they can be accessed and modified by derived contracts. It's a best practice to explicitly declare the visibility of state variables to make the code clearer and avoid potential unintended access.

C3fIb4 Non-indexed parameters in events

Low Resolved

The smart contract events include addresses as parameters, but these addresses are not marked as `indexed`. Using `indexed` for specific types of parameters, like addresses, in events facilitates efficient filtering and searching for these events on the blockchain. Without indexing, anyone who wishes to filter events by a specific address would need to retrieve and inspect every emitted event.

```
    event SubjectAdded(
        address subject
    );
    //event to indicate a trade operation
    event Trade(
        address trader,
        address subject,
```

```
        bool isBuy,
        uint256 shareAmount,
        uint256 ethAmount,
        uint256 protocolEthAmount,
        uint256 subjectEthAmount,
        uint256 holderEthAmount,
        uint256 supply,
        uint256 price
    );
    //claim of holderFee by holder
    event Claim(
        address holder,
        uint256 amount
    );
    //event to indicate that a new holder eligible for holderFee has been added
    //if oldHolder != address(0) then new holder replaced another holder who sold his
shares
    event NewEligibleHolder(
        address oldHolder,
        address holder,
        address subject,
        uint256 shares
    );
```

C3fIb6 Lack of events

Low Resolved

The `setFeeDestination` function in the smart contract modifies the `protocolFeeDestination` state variable but doesn't emit an event to log the change. Emitting events, especially for functions that change critical state variables, is vital for transparency and traceability. It allows off-chain systems and users to easily track changes, making the smart contract more reliable and user-friendly.

C3fIb7 Possible wrong fee recipient

Low Resolved

The holder fee is sent to the `protocolFeeDestination` address in the `_distributeFeeToHolders()` function if the holder is a `msg.sender`.

```
    function _distributeFeeToHolders(
        address sharesSubject, //address of the subject
        uint256 holderFee //amount to be distributed
    ) internal {
      ...
            //if the msg.sender is one of the eligible holders, his part of the fee is
sent to the protocol fee destination
            if (msg.sender != holder) {
                availableToClaim[holder] += holderFeeAmount;
            } else {
                protocolFeeDestination.call{
                    value: holderFeeAmount
                }("");
            }
      ...
```

With a lack of explicit documentation, it's hard to say whether it's an intended behavior. It looks like the fee should be sent to the caller, i.e. `msg.sender`.

Update

The team responded that this behavior is intended.

C3f1b1 Gas optimizations

Low Resolved

1. The `decimals`, `decMul`, `protocolFeePercent`, `subjectFeePercent`, `holderFeePercent` variables should be declared as constants. This will save gas on reading them. Also, they could be renamed to uppercase according to the Solidity naming conventions.

2. The queue for holders is used only for users after the `maxShareHoldersPerSubject = 600` index, there's no need to store the queue for eligible users. Queue itself is inefficient, gas efficient approach is to use array for eligible users (first 600) and a DoubleEndedQueue from OpenZepeelin for the rest.

3. Multiple reads from storage in the `_distributeFeeToHolders()` function: `maxShareHoldersPerSubject`, `subjectToEligibleHolders[sharesSubject]` . Gas could be saved with memoization.

4. Unnecessary `nonReentrant` modifier in the `initShares()` function.

5. Unnecessary call for `getPrice()` in the `Trade()` event in the `buyShares()` and `sellShares()` functions, price is already read in the `_getFees()`.

6. Multiple reads from storage in the `buyShares()` function: `subjectToHolderToIndex[sharesSubject][msg.sender]`, `sharesBalance[sharesSubject][msg.sender]`, `subjectToHolderToIndex[sharesSubject][msg.sender]`, `subjectToEligibleHolders[sharesSubject]`, `subjectToMaxIndex[sharesSubject]`.

7. Multiple reads from storage in the `sellShares()` function: `sharesBalance[sharesSubject][msg.sender]`, `subjectToEligibleHolders[sharesSubject]`.

8. The `subjectToIndexToHolder[sharesSubject][subjectToEligibleHolders[sharesSubject]]` mapping is not cleared in the `sellShares()` function.

9. The functions `getBuyPriceAfterFee()`, `getSellPriceAfterFee()`, `initShares()`, `buyShares()`, `sellShares()`, `claim()` could be declared `external` instead of `public`.

C3fIb5 Typographical errors

● Info ✓ Resolved

The smart contract contains several typographical errors that can lead to confusion and potential misinterpretation: 'whichc', 'elegible', 'is holder is already eligible...'.

C3fIb3 Unconventional naming for a constant variable

● Info ✓ Resolved

The smart contract contains a constant variable named `maxShareHoldersPerSubject`. Solidity naming conventions suppose using uppercase letters with underscores for constant variables, e.g., `MAX_SHAREHOLDERS_PER_SUBJECT`.

C3f1ba Unaddressed TODO comments

Info Resolved

Within the `buyShares()` function on line L275, there's a comment: `//@audit TODO comment` suggesting that a certain piece of code "should be moved above current else if statement" but this change was overlooked during deployment.

```
    function buyShares(address sharesSubject, uint256 amount) nonReentrant public payable
{
        ...
        //is holder is already eligible only subjectToEligibleHolderShares is updated
        if (subjectToHolderToIndex[sharesSubject][msg.sender] > 0 &&
subjectToHolderToIndex[sharesSubject][msg.sender] <= maxShareHoldersPerSubject) {
            subjectToEligibleHolderShares[sharesSubject] += amount;
        //if holder reaches balance of 1e6+ he becomes elegible or gets into queue
        } else if (
            sharesBalance[sharesSubject][msg.sender] >= decMul
        ) {
            //should be moved above current else if statement, but was overlooked during
deployment
            //if holder is already eligible or in a queue, nothing happens
            if(subjectToHolderToIndex[sharesSubject][msg.sender] > 0) {
                return;
            }
        ...
    }
```

Also, the function sellShares() has another unaddressed TODO comment.

```
    function sellShares(address sharesSubject, uint256 amount) nonReentrant public payable
{
        uint256 supply = sharesSupply[sharesSubject];
        //should be changed to supply - amount => 1e6
        require(supply > amount, "Cannot sell the last share");
        ...
    }
```

Consider refactoring the code according to the comments and removing them.

C3fIbb Lack of NatSpec documentation

Info Resolved

The smart contract lacks NatSpec documentation. NatSpec is a natural language specification format used in Solidity to write comments that describe the purpose and behavior of functions, contracts, and other code structures. Proper documentation improves code readability, aids in understanding the contract's intended functionality, and provides guidance for future developers or auditors.

6. Conclusion

1 high, 6 low severity issues were found during the audit. 6 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