HashEx
BLOCKCHAIN SECURITY

Fieres Vesting

smart contracts
preliminary audit report
for internal use only

March 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 Fieres team to perform an audit of their smart contract. The audit was conducted between 12/03/2024 and 14/03/2024.

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 at @Fiereschainorg/icodashboard-asad-cpp Github repository after the 471cbb3 commit.

Update. The Fieres team has responded to this report. The updated contracts are available in the same repository after the cd923ba commit.

2.1 Summary

Project name	Fieres Vesting
URL	https://fieres.io/
Platform	Fieres Network
Language	Solidity
Centralization level	● High
Centralization risk	● High

2.2 Contracts

Name	Address
Vesting	
Imported contracts	

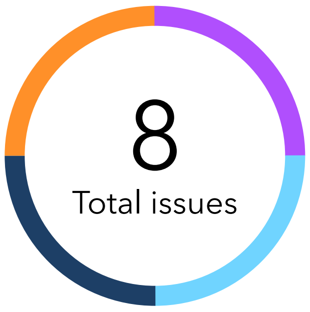
3. Project centralization risks

The project owner is responsible for depositing, and updating vesting schedule globally and individually, changing user's beneficiary address, withdrawing vested funds globally or individually.

Cc1CR15 The owner can withdraw or block vested funds

The owner of the contract has the ability to change beneficiary address for vested funds via the `changeUserClaimAddress()` function, to block all claims via `updateVestingDays()` and `updateAdminCommission()` functions, and to withdraw all vested funds at once via the `drainTokens()` function.

4. Found issues



●	High	2 (25%)
●	Medium	2 (25%)
●	Low	2 (25%)
●	Info	2 (25%)

Cc1. Vesting

ID	Severity	Title	Status
Cc1I0d	● High	Possible locked funds	Acknowledged
Cc1I12	● High	Lack of tests and documentation	Acknowledged
Cc1I0b	● Medium	Owner can block claims	Acknowledged
Cc1I14	● Medium	The getUnlockedTokenAmount() function returns incorrect number of days in corner cases	✓ Resolved
Cc1I0c	● Low	Parameter validation	✓ Resolved
Cc1I09	● Low	Gas optimizations	Partially fixed
Cc1I11	● Info	Lack of documentation (NatSpec)	Acknowledged
Cc1I0a	● Info	Lack of event	✓ Resolved

5. Contracts

Cc1. Vesting

Overview

The implemented vesting model includes fixed locking period, fixed for each user individually at the moment of vesting creation, and equally split claimable parts of the total vested amount.

Issues

Cc1I0d Possible locked funds ● High ✓ Acknowledged

The `EndDay` parameter can be updated by the owner after the initial allocation. If it has been increased, the total claimable amount would be less than user's `allocatedAmount`, meaning part of vested amount would become locked. In most extreme cases, the `EndDay` parameter may be set over the typical vested amounts, resulting in 0 return values of the `getUnlockedTokenAmount()` function.

```
function updateVestingDays(
    uint256 _vestingEndDay,
    uint256 _claimEndDay
) external onlyOwner {
    ...
    EndDay = _claimEndDay;
}

function getUnlockedTokenAmount(
    address _wallet,
    uint256 _id
) public view returns (uint256, uint256) {
    ...
    allowedAmount =
        (userVestingInfo[_wallet][_id].allocatedAmount / EndDay) *
        numberOfDays;
    ...
    }
```

Recommendation

Solidify the user's vesting terms at the moment of allocation, including the `EndDay` parameter.

Cc1I12 Lack of tests and documentation ● High ✓ Acknowledged

The project doesn't contain any tests and documentation. We urgently recommend increasing test coverage. We also suggest providing the documentation section.

Cc1I0b Owner can block claims ● Medium ✓ Acknowledged

The contract owner can block user's claim of vested funds by setting commission beyond 100% or by reverting the transfer of that commission, completely or selectively.

```
    function updateAdminCommission(uint256 _adminComm) external onlyOwner {
        adminComm = _adminComm;
        emit AdminCommissionUpdated(_adminComm);
    }

    function claimTokens(
        address _user,
        uint256 _id
    ) external nonReentrant onlyUserClaimAddress(_user) {
        ...

        (uint256 tokensToSend, uint256 numberOfDays) = getUnlockedTokenAmount(
            user.wallet,
            _id
        );
        uint256 fee = (tokensToSend * adminComm) / FEE_DIVISOR;

        payable(owner()).transfer(fee);
        payable(_user).transfer(tokensToSend - fee);
        ...
```

Recommendation

Limit maximum value of the `adminComm` parameter, use non-revertible transfer for the fee.

Cc1I14 The getUnlockedTokenAmount() function returns incorrect number of days in corner cases

● Medium ✓ Resolved

The function `getUnlockedTokenAmount()` calculates the amount a user is eligible to withdraw and the duration in days over which this amount is calculated. However, for the final day of vesting, it incorrectly returns a day count that is one higher than expected.

```
    function getUnlockedTokenAmount(address _wallet, uint256 _id)
        public
        view
        returns (uint256, uint256)
    {
        VestingInfo[] memory vestingInfo = userVestingInfo[_wallet];

        uint256 allowedAmount = 0;
        uint256 numberOfDays = 0;

        if (!allocatedUser[_wallet]) {
            return (0, 0);
        }

        if (block.timestamp >= vestingInfo[_id].nextClaimTimestamp) {
            if (vestingInfo[_id].nextClaimTimestamp != 0) {
                uint256 fromTime = block.timestamp >
 vestingInfo[_id].claimEndTimestamp ?
                    vestingInfo[_id].claimEndTimestamp - DAY_IN_SECONDS :
 block.timestamp; //
                uint256 duration = (fromTime -
                    vestingInfo[_id].nextClaimTimestamp) + DAY_IN_SECONDS;
                numberOfDays = duration / DAY_IN_SECONDS;

                allowedAmount =
                    (userVestingInfo[_wallet][_id].allocatedAmount / EndDay) *
                    numberOfDays;
            }
        }

        // allowedAmount = allowedAmount - user.claimedAmount;

        if (
            allowedAmount >
            (userVestingInfo[_wallet][_id].allocatedAmount -
                userVestingInfo[_wallet][_id].claimedAmount)
        ) {
            allowedAmount = (userVestingInfo[_wallet][_id].allocatedAmount -
                userVestingInfo[_wallet][_id].claimedAmount);
        }

        return (allowedAmount, numberOfDays);
    }
```

Proof of concept

Foundry test:

```
    function test_WrongDays() public {
        Vesting vesting = new Vesting(0, 2, 4);
        vesting.allocateForVesting{value: 1 ether}(alice, bob);

        vm.warp(5 * 86400 + block.timestamp); //first day after unlock
        (uint256 unlocked, uint256 daysNum ) = vesting.getUnlockedTokenAmount(alice, 0);
        assertEq(daysNum, 4, "days num not 4");

        vm.warp(86400 + block.timestamp);
        (unlocked, daysNum ) = vesting.getUnlockedTokenAmount(alice, 0);
        assertEq(daysNum, 5, "days is 5");

        vm.warp(86400 + block.timestamp);
        (unlocked, daysNum ) = vesting.getUnlockedTokenAmount(alice, 0);
        assertEq(daysNum, 4, "days num not 4");
    }
```

Test output:

```
$ forge test
[⠊] Compiling...
No files changed, compilation skipped

Running 1 test for test/Contract.t.sol:TestContract
[PASS] test_WrongDays() (gas: 1624971)
Test result: ok. 1 passed; 0 failed; finished in 748.83µs
```

Recommendation

Fix the calculation of days for the corner cases to return the correct number.

Cc1I0c Parameter validation

● Low ✓ Resolved

The `getUnlockedTokenAmount()` function lacks the input parameter validation. The _id parameter can cause an array indexation problem since it's not checked against the length of the `userVestingInfo[]` array.

```
    function getUnlockedTokenAmount(
        address _wallet,
        uint256 _id
    ) public view returns (uint256, uint256) {
        VestingInfo[] memory vestingInfo = userVestingInfo[_wallet];
        ...
        if (vestingInfo[_id].nextClaimTimestamp != 0) {
            ...
    }
```

Cc1I09 Gas optimizations

● Low Partially fixed

1. Excessive stored data in the `UserInfo` struct: user address is stored as a key and as a value `userInfo[addr].wallet = addr`.

2. Zero effect code in the `_allocateAmount()` function: `user.totalClaimedAmount = user.totalClaimedAmount` line doesn't update the storage.

3. Always met condition in the `claimTokens()` function: `tokensToSend > 0` is always true.

4. Unnecessary actions in the `claimTokens()` function: `nextClaimTime` calculations should be moved to the `else` section.

5. Multiple reads from storage in the `claimTokens()` function: `userVestingInfo[_user][_id].claimedAmount` variable.

6. Unnecessary reads from storage in the `getUnlockedTokenAmount()` function: `userVestingInfo[_wallet]` is read in full instead of a single array item.

7. Multiple reads from storage in the `getUnlockedTokenAmount()` function: `userVestingInfo[_wallet][_id].allocatedAmount` and `userVestingInfo[_wallet][_id].claimedAmount` variables.

8. Pointless code in the `onlyUserClaimAddress()` modifier: `userClaimAddress != address(0)` condition is always met.

9. Multiple reads from storage in the `onlyUserClaimAddress()` modifier: `userInfo[_user].userClaimAddress` is read twice.

Cc1I11 Lack of documentation (NatSpec) ● Info Acknowledged

We recommend writing documentation using NatSpec Format. This would help in development, as well as simplify user interaction with the contract (including using the block explorer).

Cc1I0a Lack of event ● Info Resolved

The `VestingDaysUpdated` event is not emitted in the constructor section, while it is emitted in the `updateVestingDays()` function with similar effect.

```
    constructor(
        uint256 _adminComm,
        uint256 _vestingEndDay,
        uint256 _claimEndDay
    ) {
        // token = IERC20(_token);
        require(
            _vestingEndDay > 0 && _claimEndDay > 0,
            " Day should not be Zero"
        );
        adminComm = _adminComm;
        vestingEndDay = _vestingEndDay * DAY_IN_SECONDS;
        claimEndDay = _claimEndDay * DAY_IN_SECONDS;
        EndDay = _claimEndDay;
    }

    function updateVestingDays(
        uint256 _vestingEndDay,
        uint256 _claimEndDay
    ) external onlyOwner {
        require(
            _vestingEndDay > 0 && _claimEndDay > 0,
            " Day should not be Zero"
        );
        vestingEndDay = _vestingEndDay * DAY_IN_SECONDS;
        claimEndDay = _claimEndDay * DAY_IN_SECONDS;
        EndDay = _claimEndDay;
        emit VestingDaysUpdated(_vestingEndDay, _claimEndDay);
    }
```

Cc2. Imported contracts

Overview

The contracts ReentrancyGuard, IERC20, Context, and Ownable are imported from the OpenZeppelin repository without modifications.

6. Conclusion

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