#HashEx
BLOCKCHAIN SECURITY

Polarys Vesting 2

smart contracts
final audit report

January 2023

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 PolarysDAC team to perform an audit of their smart contract. The audit was conducted between 02/01/2023 and 05/01/2023.

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 is available in the @PolarysDAC/polarys-contracts public GitHub repository after the commit 6b61a36.

Update. The PolarysDAC team has responded to this report, the updated code is located in the same repository after dcdedc7 commit. The audited PolarDepositContract was deployed at the Ethereum chain at address 0x061Bb29F047472eEF713B994c7320eF95E1f86aA
, TokenVesting contract was deployed to Metis Andromeda chain at address 0x64DdED4Bf12656326f3d0a40F00f973409Fea3d4.

2.1 Summary

Project name	Polarys Vesting 2
URL	https://www.polarys.io/
Platform	Metis
Language	Solidity

2.2 Contracts

Name	Address
PolarDepositContract	`0x061Bb29F047472eEF713B994c7320eF95E1f86aA`
TokenVesting	`0x64DdED4Bf12656326f3d0a40F00f973409Fea3d4`

3. Found issues

10
Total issues

●	Critical	1 (10%)
●	High	1 (10%)
●	Low	5 (50%)
●	Info	3 (30%)

C1. PolarDepositContract

ID	Severity	Title	Status
C1-01	● Low	Gas optimization	✓ Resolved
C1-02	● Low	Constructor lacks validation of input parameters	✓ Resolved
C1-03	● Info	Reentrancy guard	✓ Resolved
C1-04	● Info	Lack of documentation (NatSpec)	✓ Partially fixed
C1-05	● Info	Price calculation	✓ Acknowledged

C2. TokenVesting

ID	Severity	Title	Status
C2-01	● Critical	Zero-containing vesting	✓ Resolved
C2-02	● High	Change of the vesting schedule	✓ Resolved

C2-03	● Low	Constructor lacks validation of input parameters	✓ Resolved
C2-04	● Low	Check for 100% vesting	✓ Resolved
C2-05	● Low	Gas optimization	✓ Resolved

4. Contracts

C1. PolarDepositContract

Overview

The contract allows depositing ERC20 tokens with specifying the quantity of NFT tokens. Only users with the Deposit role and with a special signature can make a deposit. Such signatures are provided by third-party applications.

Users who have admin permission can withdraw deposited ERC20 tokens to any address.

According to the developer, the contract allows accepting payment for NFT in other chains. After payment, the contract emits events that must be processed by third-party applications. Based on these events, users will be able to receive NFT tokens on other networks.

At the same time, the audit team did not audit third-party applications.

Issues

C1-01 Gas optimization ● Low ✓ Resolved

The state variable `priceFeed` can be declared as immutable to save gas.

C1-02 Constructor lacks validation of input parameters ● Low ✓ Resolved

The contract constructor does not validate the values of the `acceptToken` and `priceAggregator` variables against non-zero value.

C1-03 Reentrancy guard ● Info ✓ Resolved

Since the audit team does not know the `_acceptToken` token, we recommend adding a nonReenrant modifier for the `depositToken()` function to prevent potential threats associated with callbacks inside the `_acceptToken` token.

C1-04 Lack of documentation (NatSpec) ● Info ✓ Partially fixed

We recommend writing documentation using NatSpec Format for the `depositNativeToken()` function and for all parameters of the `depositToken()` function. This would help in development, as well as simplify user interaction with the contract (including using the block explorer).

C1-05 Price calculation ● Info ✓ Acknowledged

The `depositNativeToken()` function calculates the quantity of tokens that will be minted on another network.

According to the current formula, the cost of one token will be about `25000000wei = 0.000000000025 ether` (or 1.2usd = 48003585 tokens at the exchange rate on January 02, 2023).

Please make sure the calculation is correct and add tests.

In addition, the `depositNativeToken()` function does not involve paying change if the user's transferred funds were not a multiple of the value of the tokens.

C2. TokenVesting

Overview

The contract is used to distribute (vest) tokens to certain users.

Contract admins with `VESTING_ROLE` can create and revoke PolarysToken vestings for any user. The admin can revoke vesting at any time. At the same time, tokens for which the release time has not yet come will be returned to the owner of the contract (to an address controlled by him).

Issues

C2-01 Zero-containing vesting ● Critical ✓ Resolved

The vesting plan can contain zero values for several months (not for the first period). In this case, users who decide to withdraw their tokens in a month with a zero value will not be able to do this, or they will be able to withdraw tokens that do not belong to them.

Case 1.

For example, the vesting plan is as follows [2000, 6000, 0, 0, 0, 0, 1000, 1000]. If the user has not withdrawn the tokens in the first two months, he will be blocked for the next four months due to zero values and the if condition (L296-298) in the `_computeReleasableAmount()` function.

```
    function _computeReleasableAmount(VestingSchedule memory vestingSchedule)
        internal
        view
        returns (uint256)
    {
        unchecked {
            uint256 currentTime = block.timestamp;
            uint256 vestedAmountByDefault = vestingSchedule.immediateVestedAmount -
vestingSchedule.released;
            if (vestingSchedule.startTimestamp > currentTime) { // vesting is not started
yet
```

```
                return vestedAmountByDefault;
            }
            uint256 monthCount = ((currentTime - vestingSchedule.startTimestamp) / 30
days) + 1;
            uint256[] memory monthUnlockPercentList =
_monthUnlockPercentSchedules[vestingSchedule.vestingGroupId];
            if (monthCount > monthUnlockPercentList.length) {   // vesting period is ended
                return vestingSchedule.amountTotal + vestedAmountByDefault;
            }

            uint256 unlockPercent = monthUnlockPercentList[monthCount-1];
            if (unlockPercent == 0) {   // cliff period
                return vestedAmountByDefault;
            }

            // vesting period
            uint256 totalPercents;
            for (uint256 index; index < monthCount; ++ index) {
                totalPercents = totalPercents + monthUnlockPercentList[index];
            }
            uint256 vestedAmount = vestingSchedule.amountTotal * totalPercents / 10000;
            vestedAmount = vestedAmount + vestedAmountByDefault;
            return vestedAmount;
        }
    }
```

Case 2.

For example, the vesting plan is as follows [2000, 6000, 0, 0, 0, 0, 1000, 1000], the `immediateVestedAmount` is zero (or less than the first vested amount) and `amountTotal` is 10000 tokens.

The user withdrew funds in the first period and now `vestingSchedule.released` is 2000. After that, the user waits for a period with zero distribution (for example 4th period) and calls the release() function with the argument `amount=polarysToken.balanceOf(address(TokenVesting))` . The calculation on L285 will return an **underflowed** result for the `release()` function. Thus, the contract balance will be transferred to the user's address.

```
    function release(bytes32 vestingScheduleId, uint256 amount)
        public
        nonReentrant
    {
        VestingSchedule storage vestingSchedule = vestingSchedules[vestingScheduleId];
        require(vestingSchedule.beneficiary != address(0), "Invalid vestingScheduleId");
        if (vestingSchedule.revoked)
            revert ScheduleRevoked();
        address beneficiary = vestingSchedule.beneficiary;
        bool isBeneficiary = msg.sender == beneficiary;
        bool isOwner = hasRole(VESTING_ROLE, msg.sender);
        if (!isBeneficiary && !isOwner) revert BeneficiaryOrOwner();
        uint256 releasableAmount = _computeReleasableAmount(vestingSchedule);
        if (releasableAmount < amount) revert NotEnoughTokens();
        vestingSchedule.released = vestingSchedule.released + amount;
        vestingSchedulesTotalAmount = vestingSchedulesTotalAmount - amount;
        polarysToken.safeTransfer(beneficiary, amount);
        emit Released(msg.sender, vestingScheduleId, amount, block.timestamp);
    }
```

Recommendation

1. We strongly recommend avoiding the unchecked mode in the `_computeReleasableAmount()` function. This will reduce the risk of incorrect calculation of tokens.

2. The zero distribution periods processing architecture needs to be redesigned.

C2-02 Change of the vesting schedule ● High ✓ Resolved

The contract admin with `VESTING_ROLE` can change any vesting schedule at any time using `setUnlockMonthSchedule()` since the admin can specify any value for the `vestingGroupIndex` parameter.

Thus already created and partially paid vestings can be changed. This can significantly affect the correctness (and honesty) of the calculation. In this case, some users will be able to receive more than 100% of the tokens (and some less than 100%).

Recommendation

Consider using a unique nonce counter for `_monthUnlockPercentSchedules` mapping.

C2-03 Constructor lacks validation of input parameters ● Low ✓ Resolved

The contract constructor does not validate the values of the `polarysToken` and `_owner` variables against non-zero value.

C2-04 Check for 100% vesting ● Low ✓ Resolved

The function `setUnlockMonthSchedule()` allows the creation of vesting plans with `totalPercents` value of less than 100%. At the same time, when creating vesting, the `createVestingSchedule()` function checks that the contract is provided with tokens in full (`amountTotal + immediateReleaseAmount`).

But if the vesting of `totalPercents` is less than 100%, then the `amountTotal` will be paid out in full only after the vesting time is completely over, even if the last periods have zero percentages. For example, for vesting month schedule [3000, 6000, 0, 0, 0, 0, 0] the user has to wait 7 months to receive 100% of the tokens.

This behavior is possible due to a non-strict check for `totalPercents` value in the `setUnlockMonthSchedule()` function.

```
    function setUnlockMonthSchedule(uint256 vestingGroupIndex, uint256[] calldata
monthUnlockPercentList) external onlyRole(VESTING_ROLE) {
        uint256[] memory monthPercentList = monthUnlockPercentList;
        uint256 monthCount = monthPercentList.length;
        uint256 totalPercents;
        for(uint256 i; i < monthCount; i = _unsafe_inc(i)) {
            unchecked {
                totalPercents = totalPercents + monthPercentList[i];
            }
            require(totalPercents <= 10000, "TotalPercent is limited to 100%");
        }
        ...
    }
```

```
    function _computeReleasableAmount(VestingSchedule memory vestingSchedule)
        internal
        view
        returns (uint256)
    {
        unchecked {
            ...
            uint256 vestedAmount = vestingSchedule.amountTotal * totalPercents / 10000;
            vestedAmount = vestedAmount + vestedAmountByDefault;
            return vestedAmount;
        }
    }
```

Recommendation

We recommend updating the `require` statment for the `totalPercents` value in the `setUnlockMonthSchedule()` function (L108):

```
require(totalPercents == 10000, "TotalPercent is limited to 100%");
```

C2-05 Gas optimization

● Low ✓ Resolved

a. Remove the unused `harhat/console.sol` import on L8.

b. The variables `startTimestamp` and `vestingGroupId` of the `VestingSchedule` structure can be packed into one storage slot by casting to `uint128`.

c. No need to create a new variable `monthPercentList` on L101 with the same values as an input parameter of the `setUnlockMonthSchedule()` function.

d. No need to create a new variable `vestingSchedule` on L268 (and read again from storage) with the same value as the local variable `memoryVestingSchedule` in the `computeReleasableAmount()` function.

5. Conclusion

1 critical, 1 high, 5 low severity issues were found during the audit. 1 critical, 1 high, 5 low issues were resolved in the update.

The contract PolarDepositContract is highly dependent on third-party applications. Users using the project have to trust the owner (as well as the project team) and be sure that the third-party applications work correctly and that vesting is created honestly.

We strongly suggest adding unit and functional tests.

This audit includes recommendations on improving the code and preventing potential attacks.

Note: no third-party application has been audited by the audit team.

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