VIDMA

HAQQ

SMART CONTRACT AUDIT

Project: Haqq Network
Date: October 5th, 2022

TABLE OF CONTENTS

SUMMARY

Vidma is pleased to present this audit report outlining our assessment of code, smart contracts, and other important audit insights and suggestions for management, developers, and users.

The audited scope included the HaqqVesting contract. The HaqqVesting allows everyone deposits ether for the specific beneficiary. Each deposit can be repaid to a beneficiary in a fixed amount of portions. The vesting period contains 24 payments with the period between each payout of 30 days. Payments can be cumulative. There can be many deposits for the same beneficiary address. It is allowed to deposit up to 5 separate portions for the same beneficiary. Any address can trigger the next payment to the beneficiary.

During the audit process, the Vidma team found several issues. A detailed summary and the current state are displayed in the table below.

Severity of the issue	Total found	Resolved	Unresolved
Critical	0 issues	0 issues	0 issues
High	0 issues	0 issues	0 issues
Medium	2 issues	2 issues	0 issues
Low	5 issues	5 issues	0 issues
Informational	4 issues	2 issues	2 issues
Total	**11 issues**	**9 issues**	**2 issues**

After evaluating the findings in this report and the final state after fixes, the Vidma auditors can state that the contracts are fully operational and secure. Under the given circumstances, we set the following risk level:

Our auditors are evaluating the initial commit given for the scope of the audit and the last commit with the fixes. This approach helps us adequately and sequentially evaluate the quality of the code. Code style, optimization of the contracts, the number of issues, and risk level of the issues are all taken into consideration. The Vidma team has developed a transparent scoring system presented below.

Severity of the issue	Resolved	Unresolved
Critical	1	10
High	0.8	7
Medium	0.5	5
Low	0.2	0.5
Informational	0	0.1

Please note that the points are deducted out of 100 for each and every issue on the list of findings (according to the current status of the issue). Issues marked as "not valid" are not subject to point deduction.

Based on the **overall result of the audit**, the Vidma audit team grants the following score:

In addition to manual check and static analysis, the auditing team has conducted a number of integrated autotests to ensure the given codebase has an adequate performance and security level.

The test results and the coverage can be found in the accompanying section of this audit report.

Please be aware that this audit does not certify the definitive reliability and security level of the contract. This document describes all vulnerabilities, typos, performance issues, and security issues found by the Vidma audit team. If the code is still under development, we highly recommend running one more audit once the code is finalized.

SCOPE OF WORK

HAQQ

Haqq is a scalable and interoperable Ethereum, built on Proof-of-Stake with fast-finality. Islamic Coin (ISLM) is a native currency of Haqq.

Within the scope of this audit, two independent auditors thoroughly investigated the given codebase and analyzed the overall security and performance of the smart contracts.

The audit was conducted from Sep 27, 2022 to Sep 29, 2022. The outcome is disclosed in this document.

The review of the fixes was made on Wednesday, October 5.

The scope of work for the given audit consists of the following contracts:

- HaqqVesting;

The source code was taken from the following **source**:
https://github.com/haqq-network/vesting-contract/tree/main/src/contracts

Initial commit submitted for the audit:
54e0feb8943f5d11d8568c0139cf4b856275f48e

Last commit reviewed by the auditing team:
aa948f9273ebe22c276e5c32a7b0715d9dd8aa5e

WORKFLOW OF THE AUDITING PROCESS

Vidma audit team uses the most sophisticated and contemporary methods and well-developed techniques to ensure contracts are free of vulnerabilities and security risks. The overall workflow consists of the following phases:

Phase 1: The research phase

Research

After the Audit kick-off, our security team conducts research on the contract's logic and expected behavior of the audited contract.

Documentation reading

Vidma auditors do a deep dive into your tech documentation with the aim of discovering all the behavior patterns of your codebase and analyzing the potential audit and testing scenarios.

The outcome

At this point, the Vidma auditors are ready to kick off the process. We set the auditing strategies and methods and are prepared to conduct the first audit part.

Phase 2: Manual part of the audit

Manual check

During the manual phase of the audit, the Vidma team manually looks through the code in order to find any security issues, typos, or discrepancies with the logic of the contract. The initial commit as stated in the agreement is taken into consideration.

Static analysis check

Static analysis tools are used to find any other vulnerabilities in smart contracts that were missed after a manual check.

The outcome

An interim report with the list of issues.

Phase 3: Testing part of the audit

Integration tests

Within the testing part, Vidma auditors run integration tests using the Truffle or Hardhat testing framework. The test coverage and the test results are inserted in the accompanying section of this audit report.

The outcome

Second interim report with the list of new issues found during the testing part of the audit process.

STRUCTURE AND ORGANIZATION OF THE FINDINGS

For simplicity in reviewing the findings in this report, Vidma auditors classify the findings in accordance with the severity level of the issues. (from most critical to least critical).

All issues are marked as “Resolved” or “Unresolved”, depending on if they have been fixed by Haqq Network or not. The issues with “Not Valid” status are left on the list of findings but are not eligible for the score points deduction.

The latest commit with the fixes reviewed by the auditors is indicated in the “Scope of Work” section of the report.

The Vidma team always provides a detailed description of the issues and recommendations on how to fix them.

Classification of found issues is graded according to 6 levels of severity described below:

Critical

The issue affects the contract in such a way that funds may be lost or allocated incorrectly, or the issue could result in a significant loss.

Example: Underflow/overflow, precisions, locked funds.

High

The issue significantly affects the ability of the contract to compile or operate. These are potential security or operational issues.

Example: Compilation errors, pausing/unpausing of some functionality, a random value, recursion, the logic that can use all gas from block (too many iterations in the loop), no limitations for locking period, cooldown, arithmetic errors which can cause underflow, etc.

Medium

The issue slightly impacts the contract's ability to operate by slightly hindering its intended behavior.

Example: Absence of emergency withdrawal of funds, using assert for parameter sanitization.

Low

The issue doesn't contain operational or security risks, but are more related to optimization of the codebase.

Example: Unused variables, inappropriate function visibility (public instead of external), useless importing of SCs, misuse or disuse of constant and immutable, absent indexing of parameters in events, absent events to track important state changes, absence of getters for important variables, usage of string as a key instead of a hash, etc.

Informational

Are classified as every point that increases onboarding time and code reading, as well as the issues which have no impact on the contract's ability to operate.

Example: Code style, NatSpec, typos, license, refactoring, naming convention (or unclear naming), layout order, functions order, lack of any type of documentation.

MANUAL REPORT

Lack of withdrawal event execution

Medium | MM – 01 | Resolved

In function *deposit()* first withdrawal for the beneficiary is made but there is no event triggered.

Recommendation:

Consider triggering *WithdrawalMade()* event for the first withdrawal in the *deposit()* function.

Incorrect passed value to the event

Medium | MM – 02 | Resolved

In function *transferDepositRights()* event *DepositMade()* is triggered. For the event param depositId value *depositsCounter[_newBeneficiaryAddress]* is passed which will be always equal to zero as it changes outside the loop.

Recommendation:

Consider passing iterator depositId in the event L196 instead to keep the valid data.

Lack of zero value check

Low | ML – 01 | Resolved

In function *deposit()* the ether is deposited for the passed as input param address. But there is no check if the passed address of beneficiary or sent eth isn't zero value.

Recommendation:

Consider adding a require statement to check if the *_beneficiaryAddress* value isn't zero address and msg.value isn't zero value.

Useless arithmetic operation execution

Low | ML – 02 | Resolved

In the function *deposit()* variable *sumPaidAlready* assigns to the number of the first withdrawal made. When *deposit()* is called multiple times for one beneficiary data is stored for the unique depositId identifier so *sumPaidAlready* always will be equal to zero when *deposit()* function is executed.

There is a possibility to avoid the addition of the previous *sumPaidAlready* with the *firstWithdrawalAmount* variable as *sumPaidAlready* will be always zero:

```
deposits[_beneficiaryAddress][depositId].sumPaidAlready =
firstWithdrawalAmount;
```

Recommendation:

Consider removing useless addition to decrease gas usage on the function execution.

Lack of delete operator

Low | ML – 03 | Resolved

In function *transferDepositRights()* assigns zero value to deposits and *depositsCounter* mappings. It is more efficient to use delete operator here to save gas usage on the function execution.

Recommendation:

Consider using delete operator to clear *deposits* and *depositsCounter* mappings.

L193:

```
delete deposits[msg.sender][depositId];
```

L203:

```
delete depositsCounter[msg.sender];
```

Too many storage references

Low | TL – 01 | Resolved

The *deposit()* function of the HaqqVesting Smart Contract uses next block of code:

```
deposits[_beneficiaryAddress][depositId].sumPaidAlready =
deposits[_beneficiaryAddress][depositId].sumPaidAlready +
      firstWithdrawalAmount;
deposits[_beneficiaryAddress][depositId].timestamp =
block.timestamp;
deposits[_beneficiaryAddress][depositId].sumInWeiDeposited =
msg.value;
```

It is could be optimized to the next state:

```
deposits[_beneficiaryAddress][depositId] = Deposit({
timestamp: block.timestamp,
sumInWeiDeposited: msg.value,
sumPaidAlready: firstWithdrawalAmount
});
```

Also getting values from memory variables will be `cheaper ` than using variable from storage. For example, in event emitting:

```
emit DepositMade(
_beneficiaryAddress,
depositsCounter[_beneficiaryAddress],
deposits[_beneficiaryAddress][depositId].timestamp, //! could be
optimized to block.timestamp
deposits[_beneficiaryAddress][depositId].sumInWeiDeposited, //! to
msg.value
msg.sender
);
```

Full function code:

```
/// @dev Function to make a new deposit.
/// @param _beneficiaryAddress address that will receive payments
from this deposit
function deposit(address _beneficiaryAddress)
external
payable
nonReentrant
returns (bool success)
{
        // new deposit id for this deposit
        uint256 depositId = ++depositsCounter[_beneficiaryAddress];
        require(
            depositId <= MAX_DEPOSITS,
            "Max deposit number for this address reached"
        );

        // make the first withdrawal
        // if beneficiary address can not receive ETH, there will be
no deposit for this address
        uint256 firstWithdrawalAmount = msg.value /
```

```
NUMBER_OF_PAYMENTS;
        (bool sent, ) = payable(_beneficiaryAddress).call{
            value: firstWithdrawalAmount
        }("");
        require(sent, "Failed to send Ether");

        // after the first withdrawal succeeded, make records to
this deposit:

        deposits[_beneficiaryAddress][depositId] = Deposit({
            timestamp: block.timestamp,
            sumInWeiDeposited: msg.value,
            sumPaidAlready: firstWithdrawalAmount
        });

        // emit event:

        emit DepositMade(
            _beneficiaryAddress,
            depositId,
            block.timestamp,
            msg.value,
            msg.sender
        );

return true;
}
```

Recommendation:

Consider gas economy.

Optimization of total payouts functionality

Low | TL – 02 | Resolved

The totalPayoutsUnblocked() function of the HaqqVesting Smart Contract declares depositAge but it is used only once, so it is could be already moved to payouts count calculating. Also if-branching could be missed:

```
/// @dev Total payouts unlocked in the elapsed time.
/// @dev One payment is unlocked immediately
function totalPayoutsUnblocked(
    address _beneficiaryAddress,
    uint256 _depositId
) public view returns (uint256) {
        require(
            deposits[_beneficiaryAddress][_depositId].timestamp > 0,
            "No deposit with this ID for this address"
        );

        uint256 totalPayoutsUnblocked_ = ((block.timestamp -
            deposits[_beneficiaryAddress][_depositId].timestamp) /
            TIME_BETWEEN_PAYMENTS) + 1;

        return
            totalPayoutsUnblocked_ > NUMBER_OF_PAYMENTS
                ? NUMBER_OF_PAYMENTS
                : totalPayoutsUnblocked_;
}
```

Recommendation:

Consider optimization.

Order of Layout

Informational | MI – 01 | Unresolved

The HaqqVesting contract elements should be grouped and ordered in the following way:

- Pragma statements;
- Import statements;
- Interfaces;
- Libraries;
- Contract.

Inside each contract, library or interface, use the following order:

- Library declarations (using statements);
- Constant variables;
- Type declarations;
- State variables;
- Events;
- Modifiers;
- Functions.

Ordering helps readers to navigate the code and find the elements more quickly.

Recommendation:

Consider changing the order of layout according to solidity documentation: Order of Layout.

Order of functions

Informational | MI – 02 | Unresolved

The functions in contract HaqqVesting are not grouped according to their visibility and order.

Functions should be grouped according to their visibility and ordered in the following way:

- constructor;
- receive function (if exists);
- fallback function (if exists);
- external;
- public;
- internal;
- private.

Ordering helps readers navigate across the code, identify which functions they can call and find the constructor and fallback definitions easier.

Recommendation:

Consider changing functions order according to solidity documentation: Order of Functions.

Unused local variable

Informational | MI – 03 | Resolved

In functions *deposit()* and *withdraw()* of HaqqVesting contract returned bytes memory data param of *call()* function execution is not used.

Recommendation:

Consider silence compile warning by deleting or commenting on unused variables L61 and L173.

Lack of NatSpec comments

Informational | MI – 04 | Resolved

The HaqqVesting contract is not fully covered by NatSpec. There are some functions that miss/not fully cover tag param (*calculateAvailableSumForAllDeposits()*, *amountToWithdrawNow()*, *totalPayoutsUnblocked()*)

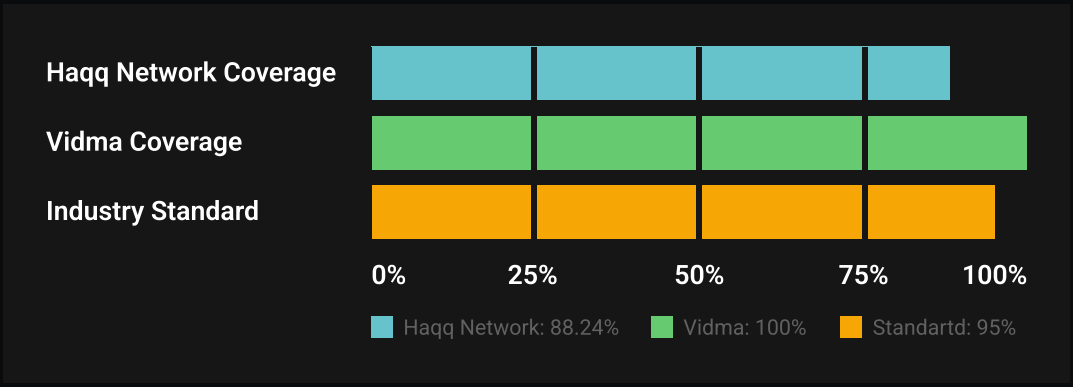
Recommendation:

Consider covering all functions with qualitative and fully NatSpec comments.

TEST RESULTS

To verify the security of contracts and the performance, a number of integration tests were carried out using the Truffle testing framework.

In this section, we provide both tests written by Haqq Network and tests written by Vidma auditors.



It is important to note that Vidma auditors do not modify, edit or add tests to the existing tests provided in the Haqq Network repository. We write totally separate tests with code coverage of a minimum of 95% to meet industry standards.

Tests written by Haqq Network

Test Coverage

File	% Stmts	% Branch	% Funcs	% Lines
contracts\	88.57	56.67	85.71	88.89
HaqqVesting.sol	88.57	56.67	85.71	88.89
All Files	**88.57**	**56.67**	**85.71**	**88.89**

Test Results

```
Contract: HaqqVesting
  ✓ Should read number of withdrawals allowed for each deposit (52ms)
balances before deposit: contract 0 beneficiary
10000000000000000000000
deposit amount: 1500000000000000000
balances after deposit: contract 14375000000000000000 beneficiary
10000625000000000000000
  ✓ Should make deposit (134ms)
  ✓ Should be able make 5 deposits (271ms)
  ✓ Should not be able make 6 deposits (283ms)
  ✓ Should transfer deposit ownership (156ms)
Testing withdrawals:
Number of payments for one deposit: 24  (one already made)
Time between payments: 2592000 seconds
contract balance before: 14375000000000000000
beneficiary address balance before: 10000625000000000000000
(1)
contract balance after withdrawal: 13750000000000000000 beneficiary:
10001250000000000000000
(2)
contract balance after withdrawal: 13125000000000000000 beneficiary:
10001875000000000000000
```

```
(3)
contract balance after withdrawal: 12500000000000000000 beneficiary:
100025000000000000000
(4)
contract balance after withdrawal: 11875000000000000000 beneficiary:
100031250000000000000
(5)
contract balance after withdrawal: 11250000000000000000 beneficiary:
100037500000000000000
(6)
contract balance after withdrawal: 10625000000000000000 beneficiary:
100043750000000000000
(7)
contract balance after withdrawal: 10000000000000000000 beneficiary:
100050000000000000000
(8)
contract balance after withdrawal: 9375000000000000000 beneficiary:
100056250000000000000
(9)
contract balance after withdrawal: 8750000000000000000 beneficiary:
100062500000000000000
(10)
contract balance after withdrawal: 8125000000000000000 beneficiary:
100068750000000000000
(11)
contract balance after withdrawal: 7500000000000000000 beneficiary:
100075000000000000000
(12)
contract balance after withdrawal: 6875000000000000000 beneficiary:
100081250000000000000
(13)
contract balance after withdrawal: 6250000000000000000 beneficiary:
100087500000000000000
(14)
contract balance after withdrawal: 5625000000000000000 beneficiary:
100093750000000000000
(15)
contract balance after withdrawal: 5000000000000000000 beneficiary:
100100000000000000000
(16)
contract balance after withdrawal: 4375000000000000000 beneficiary:
100106250000000000000
```

```
(17)
contract balance after withdrawal: 3750000000000000000 beneficiary:
1001125000000000000000
(18)
contract balance after withdrawal: 3125000000000000000 beneficiary:
1001187500000000000000
(19)
contract balance after withdrawal: 2500000000000000000 beneficiary:
1001250000000000000000
(20)
contract balance after withdrawal: 1875000000000000000 beneficiary:
1001312500000000000000
(21)
contract balance after withdrawal: 1250000000000000000 beneficiary:
1001375000000000000000
(22)
contract balance after withdrawal: 625000000000000000 beneficiary:
1001437500000000000000
(23)
contract balance after withdrawal: 0 beneficiary:
1001500000000000000000
  ✓ Should withdraw deposit (2203ms)
  ✓ Should not be able to transfer deposit ownership to alread
    used account (177ms)

7 passing (5s)
```

Tests written by Vidma auditors

Test Coverage

File	% Stmts	% Branch	% Funcs	% Lines
contracts\	100.00	100.00	100.00	100.00
HaqqVesting.sol	100.00	100.00	100.00	100.00
All Files	**100.00**	**100.00**	**100.00**	**100.00**

Test Results

```
Contract: HaqqVesting
  Deployment
    ✓ check NUMBER_OF_PAYMENTS
    ✓ check TIME_BETWEEN_PAYMENTS
    ✓ check MAX_DEPOSITS
  Functions
    deposit
      ✓ should fail if beneficiary has zero address
      ✓ should fail if deposited amount is zero
      ✓ should fail if beneficiary cannot receive first
        payment (59ms)
      ✓ should fail if it's more than fifth deposit (155ms)
      ✓ should deposit correctly (51ms)
    totalPayoutsUnblocked
      ✓ should fail if selected deposit does not exist
      ✓ should return total payouts correctly
    amountForOneWithdrawal
      ✓ should return 0 if selected deposit does not exist
      ✓ should return amount for one withdrawal correctly
    amountToWithdrawNow
      ✓ should return 0 if all rewards have already withdrawn (47ms)
      ✓ should return amount for second withdrawal correctly (39ms)
```

```
      ✓ should calculate amount for withdrawal correctly if
        some amount has already withdrawn (72ms)
    calculateAvailableSumForAllDeposits
      ✓ should return 0 if beneficiary has no deposits
      ✓ should return calculated available amount correctly (153ms)
    withdraw
      ✓ should fail if selected beneficiary does not exist
      ✓ should fail if withdrawal amount is zero (66ms)
      ✓ should fail if beneficiary cannot receive ether (47ms)
      ✓ should withdraw ether correctly (49ms)
    transferDepositRights
      ✓ should fail if caller does not have deposits
      ✓ should fail if selected beneficiary has already
        deposits (38ms)
      ✓ should transfer deposit rights correctly (74ms)
  Reentrancy
    ✓ should fail withdraw (72ms)
    ✓ should fail deposit
  Test Cases
    ✓ any address can make a deposit (122ms)
    ✓ any address can trigger the next payment to the
      beneficiary (128ms)
    ✓ multiple deposits (137ms)
    ✓ multiple withdraws (all eth should be accumulated in 1 year
      11 months) (431ms)
    ✓ first payment is made immediately while deposit is called

31 passing (2s)
```

We are delighted to have a chance to work with the Haqq Network team and contribute to your company's success by reviewing and certifying the security of your smart contracts.

The statements made in this document should be interpreted neither as investment or legal advice, nor should its authors be held accountable for decisions made based on this document.

Vidma is a security audit company helping crypto companies ensure their code and products operate safely and as intended, enabling founders to sleep soundly at night. We specialize in auditing DeFi protocols, layer one protocols, and marketplace solutions. Our team consists of experienced and internationally trained specialists. Our company is based in Ukraine, known for its strong engineering, cryptography, and cybersecurity culture.

Website: **vidma.io**
Email: **security@vidma.io**