May 21, 2019

Audit Report

GATH3R TOKEN AND CROWD SALE CONTRACTS

AUTHOR: STEFAN BEYER – CRYPTRONICS.IO

Consutling Cryptronics Oü, Sepapaja 6, Tallinn 15551, Estonia, Registry code: 14481983
VAT number: EE102067569

Cryptronics

TABLE OF CONTENTS

DISCLAIMER - 3 -

INTRODUCTION - 4 -

PURPOSE OF THIS REPORT - 4 -
CODEBASE SUBMITTED TO THE AUDIT - 4 -
METHODOLOGY - 5 -

SMART CONTRACT OVERVIEW - 6 -

TOKEN CONTRACT - 6 -
PRE-SALE CONTRACT - 6 -
CROWD SALE CONTRACT - 6 -
ASSORTED LIBRARIES - 6 -

SUMMARY OF FINDINGS - 7 -

ISSUES ENCOUNTERED - 8 -

HIGH SEVERITY ISSUES - 8 -
MEDIUM SEVERITY ISSUES - 8 -
LOW SEVERITY ISSUES - 8 -
ADDITIONAL RECOMMENDATIONS - 8 -
EXPRESS HARDCAP IN ETH - 8 -

SECURITY AUDIT BREAKDOWN - 9 -

REENTRANCY AND RACE CONDITIONS RESISTANCE - 9 -
DESCRIPTION - 9 -
AUDIT RESULT - 9 -
UNDER-/OVERFLOW PROTECTION - 9 -
DESCRIPTION - 9 -
AUDIT RESULT - 10 -
TRANSACTION ORDERING ASSUMPTIONS - 10 -
DESCRIPTION - 10 -
AUDIT RESULT - 10 -
TIMESTAMP DEPENDENCIES - 10 -
DESCRIPTION - 10 -
AUDIT RESULT - 10 -
DENIAL OF SERVICE ATTACK PREVENTION - 10 -
DESCRIPTION - 11 -

AUDIT RESULT - 11 -
BLOCK GAS LIMIT - 11 -
DESCRIPTION - 11 -
AUDIT RESULT - 11 -
STORAGE ALLOCATION PROTECTION - 11 -
DESCRIPTION - 11 -
AUDIT RESULT - 11 -
COMMUNITY AUDITED CODE - 11 -
DESCRIPTION - 11 -
AUDIT RESULT - 12 -

GAS USAGE ANALYSIS - 13 -

DESCRIPTION - 13 -
AUDIT RESULT - 13 -

DISCLAIMER

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

INTRODUCTION

PURPOSE OF THIS REPORT

The author of this report has been engaged to perform an audit ERC-20 token and crowd sale smart contracts of the Gath3r project (https://gath3r.io/)

The objectives of the audit are as follows:

1. Determine correct functioning of the contract, in accordance with the ERC-20 specification and the project whitepaper.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine contract bugs, which might lead to unexpected behavior.
4. Analyze, whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents the summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

CODEBASE SUBMITTED TO THE AUDIT

The smart contract code has been provided by the developers in form of zip file containing the source code.

The files relevant to the audit are listed below with their SHA256 hash values:

```
SHA256(Crowdsale.sol)=
5e42cdc6f9836907c40632bd70fb027670f2ab8c9c3c068318bd0d10e2fdf502
SHA256(CrowdsaleController.sol)=
180827e9f1c5f61538d9a315b0db9a512eb5f2d2ea08dbf0f30ae439dcf65c2d
SHA256(Gather_coin.sol)=
c766fc91231a01a220949c9c0e2de5bcc75725efc325c991eb31507bbf840896
SHA256(Migrations.sol)=
88eb11d021d7fd798469271d67c3cbadd4f3f055fab5c602a73930901783cbe9
SHA256(Ownable.sol)=
df3154f9faba8d203a4ab5b8cc0d28613869b2fcad884a4c1e240361bb2d3455
SHA256(Pauseble.sol)=
876e628e667a47093aa310a60d29b02777db185f10d0927926ca6eb0d1bf0bf4
SHA256(Presale.sol)=
cdc32bca083d955baa9f08d47f86ab290c53c4a6d28d723f97c3bf7da679ae3b
SHA256(PresaleController.sol)=
aaf4c786538486769f878d9f21eba3913ac876736b450c87cd7fa17f6452f1ae
SHA256(SafeMath.sol)=
b1027b3c8df2f780ff3d414bab097d782b7efff7545b1c7d961507362ca49248
SHA256(TokenController.sol)=
faead93bdd2e476a25996a070898bd760efb4fdcc32a890f6c19c77ca01e4820
SHA256(TokenHolder.sol)=
d7f7e1cc76bc0e6cf2445453e4b903de7bcfba930b606e685e9dbfa93b98cf75
```

SHA256(multiowned.sol)=
07ab0d32a5a74f59d3b8c35d10511f87768f7621c85b2609376392d100374a01
SHA256(oraclizeAPI.sol)=
ead841cb26259ba5beae3247fe09b419ea569cc8781cce7a5a32e9b20dc86464

METHODOLOGY

The audit has been performed in the following steps:

1. Gaining an understanding of the contract's intended purpose by reading the available documentation.
2. Automated scanning of the contract with static code analysis tools for security vulnerabilities and use of best practice guidelines.
3. Manual line by line analysis of the contracts source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    - Reentrancy analysis
    - Race condition analysis
    - Front-running issues and transaction order dependencies
    - Time dependencies
    - Under- / overflow issues
    - Function visibility Issues
    - Possible denial of service attacks
    - Storage Layout Vulnerabilities
4. Report preparation

SMART CONTRACT OVERVIEW

TOKEN CONTRACT

The submitted token contract is a fungible token following the ERC-20 defined in EIP-20.

The token contract extends the functionality of the ERC-20 standard by adding the following additional functionality:

- Token contract ownership
- Transfers can be paused
- Whitelist support for addresses that can transfer tokens, even when transfers are paused
- Minting support, until the owner declares minting finished
- After a certain date, new tokens can be minted again exactly once

PRE-SALE CONTRACT

The pre-sale contract allows investors to whitelisted investors before the crowd sale has started. The contract allows for different bonuses for investors.

CROWD SALE CONTRACT

The crowd sale contract fairly straightforward. Bonuses depend on the amount invested. The hardcap is set in USD. In order to calculate the correct number of tokens and caps, the contract uses the Oraclize API (http://www.oraclize.it/).

ASSORTED LIBRARIES

Library contracts are included to support the following functionalities:

- Safe arithmetic operations to avoid overflow and underflow issues
- Contract ownership
- Pausable contracts
- Multiowner proxies for token and crowd sale contracts. These contracts can be used to provide multisignature approval for certain transactions.

SUMMARY OF FINDINGS

The contracts provided for this audit are of good quality.

Community audited code seems to have been reused whenever possible. A safe math library is used almost everywhere to prevent overflow and underflow issues.

No reentrancy attack vectors have been found and precautions have been taken to avoid uninitialized storage pointers that may lead to overwriting storage. A safe math library is used throughout.

Gas usage is reasonable for this type of contract.

ISSUES ENCOUNTERED

HIGH SEVERITY ISSUES

No high severity issues have been found.

MEDIUM SEVERITY ISSUES

No medium severity issues have been found.

LOW SEVERITY ISSUES

No low severity issues have been found.

ADDITIONAL RECOMMENDATIONS

EXPRESS HARDCAP IN ETH

The hardcap of the crowd sale is expressed in USD. To calculate whether the cap has been reached, the Oraclize API is used to fetch an ETH to USD conversion rate. This is probably due to ETH volatility but introduces unnecessary complexity into the contract and an additional cost.

Whilst this is sometimes implemented this way, we recommend using an ETH or token hardcap to simplify the smart contract.

SECURITY AUDIT BREAKDOWN

REENTRANCY AND RACE CONDITIONS RESISTANCE

DESCRIPTION

Reentrancy vulnerabilities consist in unexpected behavior, if a function is called various times before execution has completed.

Let's look at the following function, which can be used to withdraw the total balance of the caller from a contract:

```
1.  mapping(address => uint) private balances;
2.
3.  function payOut() {

4.      require(msg.sender.call.value(balances[msg.sender])());
5.      balances[msg.sender] = 0;

6.  }
```

The *call.value()* invocation causes contract external code to be executed. If the caller is another contract, this means that the contracts fallback method is executed. This may call *payOut()* again, before the balance is set to 0, thereby obtaining more funds than available.

AUDIT RESULT

No reentrancy issues have been found in the contract. The transfer() function is used for all ether transfers, imposing a gas limit and preventing recursive calls, and care has been taken in the order of calls.

UNDER-/OVERFLOW PROTECTION

DESCRIPTION

Balances are usually represented by unsigned integers, typically 256-bit numbers in Solidity. When unsigned integers overflow or underflow, their value changes dramatically. Let's look at the following example of a more common underflow (numbers shortened for readability):

```
  0x0003
- 0x0004
--------
  0xFFFF
```

It's easy to see the issue here. Subtracting 1 more than available balance causes an underflow. The resulting balance is now a large number.

Also note, that in integer arithmetic division is troublesome, due to rounding errors.

AUDIT RESULT

The contracts avoid overflow and underflow issues by employing a safe math library for most arithmetic operations. In the very few places the safe math library is not used, additional checks are in place.

TRANSACTION ORDERING ASSUMPTIONS

DESCRIPTION

Transactions enter a pool of unconfirmed transactions and maybe included in blocks by miners in any order, depending on the miner's transaction selection criteria, which is probably some algorithm aimed at achieving maximum earnings from transaction fees, but could be anything. Hence, the order of transactions being included can be completely different to the order in which they are generated. Therefore, contract code cannot make any assumptions on transaction order.

Apart from unexpected results in contract execution, there is a possible attack vector in this, as transactions are visible in the mempool and their execution can be predicted. This maybe an issue in trading, where delaying a transaction may be used for personal advantage by a rogue miner. In fact, simply being aware of certain transactions before they are executed can be used as advantage by anyone, not just miners.

AUDIT RESULT

Transactions are kept as simple as possible and care has been taken not to assume a specific order of invocation.

TIMESTAMP DEPENDENCIES

DESCRIPTION

Timestamps are generated by the miners. Therefore, no contract should rely on the block timestamp for critical operations, such as using it as a seed for random number generation. Consensys give a 15 seconds rule their guidelines, which states that it is safe to use *block.timestamp,* if your time depending code can deal with a 15 second variation.

AUDIT RESULT

The only use of block timestamps in the provided contract code complies with the 15 second rule.

DENIAL OF SERVICE ATTACK PREVENTION

DESCRIPTION

Denial of Service attacks can occur when a transaction depends on the outcome of an external call. A typical example of this some activity to be carried out after an Ether transfer. If the receiver is another contract, it can reject the transfer causing the whole transaction to fail.

AUDIT RESULT

The contracts avoid DoS attacks of this type. There are no external calls that may be reverted and cause a DoS issue.

BLOCK GAS LIMIT

DESCRIPTION

Contract transactions can sometimes be forced to always fails by making them exceed the maximum amount of gas that can be included in a block. The classic example of this is explained in this explanation of an auction contract. Forcing the contract to refund many small bids, which are not accepted, will bump up the gas used and, if this exceeds the block gas limit, the whole transaction will fail.

The solution to this problem is avoiding situations in which many transaction calls can be caused by the same function invocation, especially if the number of calls can be influenced externally.

AUDIT RESULT

The contracts have no block gas limit issues. Any iterations over arrays of variable length can be split into various transations.

STORAGE ALLOCATION PROTECTION

DESCRIPTION

Storage management in Solidity can be complicated. Declarations of structs inside the scope of a function default to storage pointers. It is therefore easy to end up with and uninitialized storage pointer, pointing to address 0, instead of declaring a new struct.

Writing to this pointer then causes storage to be overwritten unintentionally.

AUDIT RESULT

No issues related to his have been found during the audit.

COMMUNITY AUDITED CODE

DESCRIPTION

It always best to re-use community audited code when available, such as the code provided by Open Zeppelin.

AUDIT RESULT

The contracts seem to be based on various sources of community audited code, including Open Zeppelin and MixBytes (https://github.com/mixbytes/solidity/tree/master/contracts).

GAS USAGE ANALYSIS

DESCRIPTION

Gas usage of smart contracts is very important. Gas is charged for each operation that alters state, i.e. a write transaction. In contrast, read-only queries can be processed by local nodes and therefore do not have an associated cost.

Excessive gas usage may make contracts unusable in practice, in particular in times of network congestion when the gas price has to be increased to incentivize miners to prioritize transactions.

Furthermore, issues with excessive gas usage can lead to exceeding the block gas limit preventing transactions from completing. This is particularly dangerous in the case of executing code in unbounded loops, for example iterating over a variable size array. If the size of the array can be influenced by a public contract call, this can be used to create Denial of Service Attacks.

For these reasons, the present smart contract audit includes a gas usage analysis performed in two steps:

1. The code has been analyzed using automated gas estimation tools that return a relatively accurate estimate of the gas usage of each function.
2. As automated, gas estimation has its limits, a manual line by line analysis for gas related issues has also been performed.

AUDIT RESULT

It is obvious that care has been taken to implement all functions as compact and gas efficiently as possible.

One source of gas related cost is the use of Oraclize for obtaining the current ETH price in the crowd sale contract. This could be avoided by expressing the hardcap in ETH.

In general, gas usage is reasonable.