June 22, 2018

Audit Report

XCHNG 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

Cryptronics

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.

Cryptronics

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 XCHNG project (https://www.xchng.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 access to the projects public source code repository:

https://github.com/kochavalabs/xchng-solidity

Commit number `bac58a71bc4338195383020f7bb3bde0500c1cb3` was the latest version at the time of the audit and has been analyzed for this report.

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 and ERC-223 specification defined in EIP-20 and ERC-223 respectively.

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

- Pausable token support
- Token contract ownership
- A burn functionality

CROWD SALE CONTRACT

The crowd sale contract implements a Dutch auction. The initial price is reduced according to a decay calculation as time progresses since the auction start. The sale finalizes once a specific target price is reached.

The minimum bid of 1 ETH and the lock time after the auction end (7 days) are hardcoded into the contract. All other parameters can be defined in the constructor or set by a specific *setup()* function.

TOKEN DISTRIBUTION CONTRACT

A token distribution contract is implemented for bulk distributing unclaimed tokens. The distribution contract interacts with the Dutch auction contract.

ASSORTED LIBRARIES

Library contracts are included to support the following functionalities:

- Safe arithmetic operations to avoid overflow and underflow issues
- Contract ownership
- Pausable contracts
- Whitelist support

SUMMARY OF FINDINGS

The contracts provided for this audit are of exceptional quality. No security issues have been detected.

Community audited code seems to have been reused whenever possible, although several sources have been combined and integrated with updated code. A safe math library is used throughput the code 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.

Gas usage is very reasonable for this type of contract.

The Dutch auction crowd sale contract depends on the price to decay to a certain minimum. The decay rate depends on two constructor variables: *price_constant* and *price_exponent*. Thus, the auction duration and the actual conversion of the investment into tokens depend on the correct configuration of these parameters. The team is aware of this and plans to disclose these parameters on the website. Correctness of these parameters and the maximum auction duration can be independently verified after the deployment of the contract, but not at the time of this audit.

AUDIT FINDING

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 XCHNG contracts avoid overflow and underflow issues by employing a safe math library for all arithmetic operations.

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

XCHNG 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 30 seconds and a 12 minutes rules in their guidelines, which states that it is safe to use *block.timestamp,* if your time depending code can deal with a 30 second or 12 minute time variation, depending on the intended use.

AUDIT RESULT

The only use of block timestamps int the provided contract code complies with the 12 minutes rule. The block time stamp is used to ensure that the waiting period since the crowd sale end date has passed. As this is 7 days, the use of block timestamps is considered safe in this case.

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 XCHNG 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 only potentially unbounded iteration over variable-sized arrays is in the *distribute()* function of the *DutchAuctionDistributor* function:

```
34    function distribute(address[] _addresses) public {
35        for (uint32 i = 0; i < _addresses.length; i++) {
36            if (auction.bids(_addresses[i]) > 0) {
37                auction.proxyClaimTokens(_addresses[i]);
38            }
39        }
40    }
```

In theory the *_addresses* array could grow large enough for the block gas limit to be reached. In praxis, this is mitigated due to the minimum investment of 1 ETH.

Furthermore, in case of a block gas limit issue, the intended behavior can be replicated using off-chain code calling *proxyClaimTokens()* individually.

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 XCHNG contracts seem to be based on various sources of community audited code, including Open Zeppelin and the ERC-223 reference implementation. The code has been integrated with XCHNG specific code and compiler version related updates.

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

AUTOMATED ANALYSIS

The following is the output of the automated gas usage analysis:

```
======= DutchAuction.sol:DutchAuction =======
Gas estimation:
construction:
   145638 + 1870800 = 2016438
external:
   fallback: infinite
   MIN_BID():    876
   WAITING_PERIOD(): 612
   addAddressToWhitelist(address):    22798
   addAddressesToWhitelist(address[]):infinite
   auction_end_time():   834
   auction_funds_claimed():   922
   auction_received_wei():    658
   auction_start_block():394
   auction_start_time(): 482
   bid():    infinite
   bids(address):    884
   claimTokens():    infinite
   final_price():    438
   finalizeAuction():infinite
   missingFundsToEndAuction():    infinite
   num_tokens_auctioned():    416
   owner():  948
   pause():  22299
   paused(): 822
   price():  infinite
   price_constant(): 966
   price_exponent(): 508
   price_start():    614
   proxyClaimTokens(address): infinite
   removeAddressFromWhitelist(address):    22531
   removeAddressesFromWhitelist(address[]):    infinite
   setup(address,address):    infinite
   stage():  1100
   startAuction():   63685
   token():  1234
   token_multiplier():   548
   token_wallet():   552
   transferOwnership(address):    23313
```

```
   unpause():    22098
   waitingPeriodOver():  infinite
   wallet_address(): 1168
   whitelist(address):   1119
internal:
   calcTokenPrice(): infinite

======= DutchAuctionDistributor.sol:DutchAuctionDistributor
=======
Gas estimation:
construction:
   21400 + 190200 = 211600
external:
   auction():    464
   distribute(address[]): infinite

======= ERC20Interface.sol:ERC20Interface =======
Gas estimation:

======= ERC223Interface.sol:ERC223Interface =======
Gas estimation:

======= ERC223ReceivingContract.sol:ERC223ReceivingContract
=======
Gas estimation:

======= Migrations.sol:Migrations =======
Gas estimation:
construction:
   20462 + 152000 = 172462
external:
   last_completed_migration():    416
   owner():  486
   setCompleted(uint256): 20544
   upgrade(address): infinite

======= Ownable.sol:Ownable =======
Gas estimation:
construction:
   20443 + 131400 = 151843
external:
   owner():  442
   transferOwnership(address):    22587

======= Pausable.sol:Pausable =======
Gas estimation:
construction:
   40834 + 236800 = 277634
external:
   owner():  508
   pause():  21859
   paused(): 514
```

```
   transferOwnership(address):    22653
   unpause():    21812

======= SafeMath.sol:SafeMath =======
Gas estimation:
construction:
   116 + 15200 = 15316
internal:
   add(uint256,uint256):  infinite
   div(uint256,uint256):  infinite
   mul(uint256,uint256):  infinite
   sub(uint256,uint256):  infinite

======= StandardToken.sol:StandardToken =======
Gas estimation:
construction:
   942 + 906400 = 907342
external:
   allowance(address,address):    838
   approve(address,uint256):  22331
   balanceOf(address):    647
   balances(address):598
   totalSupply():    416
   transfer(address,uint256): infinite
   transfer(address,uint256,bytes):   infinite
   transferFrom(address,address,uint256):  infinite

======= Whitelist.sol:Whitelist =======
Gas estimation:
construction:
   20747 + 470200 = 490947
external:
   addAddressToWhitelist(address):    22380
   addAddressesToWhitelist(address[]):infinite
   owner():  508
   removeAddressFromWhitelist(address):   22355
   removeAddressesFromWhitelist(address[]):   infinite
   transferOwnership(address):    22697
   whitelist(address):   657

======= XchngToken.sol:XchngToken =======
Gas estimation:
construction:
   86211 + 1362400 = 1448611
external:
   allowance(address,address):    1014
   approve(address,uint256):  22663
   balanceOf(address):    757
   balances(address):620
   burn(uint256):    infinite
   decimals():   316
   name():   infinite
```

```
owner():  684
pause():  21935
paused(): 618
symbol(): infinite
totalSupply():    438
transfer(address,uint256): infinite
transfer(address,uint256,bytes):   infinite
transferFrom(address,address,uint256): infinite
transferOwnership(address):    22917
unpause():    21844
```

As can be seen, gas usage of all functions, for which a numerical result was returned, is very reasonable.

Infinite gas estimates are due to the limitations of automated gas analysis. These functions have been analyzed manually.

MANUAL ANALYSIS

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

Gas usage is very reasonable.

SECURITY ISSUES

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

SMALL CHANGES TO SAFE MATH LIBRARY

Openzeppelin's Safe Math library has been simplified. The employed version is perfectly fine but could be simplified very slightly. See: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol

Furthermore, *revert* is used instead of *assert*. The recommended use of these keyword suggests the use of *assert* for this type of error checking. See the following links:

https://medium.com/blockchannel/the-use-of-revert-assert-and-require-in-solidity-and-the-new-revert-opcode-in-the-evm-1a3a7990e06e

http://solidity.readthedocs.io/en/v0.4.24/control-structures.html#error-handling-assert-require-revert-and-exceptions