September 07, 2018

Audit Report

TOLAR 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

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 Tolar project (https://www.tolar.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 public source code repository at:
https://github.com/runningbeta/tolar/tree/fcc46608b4ffd2cf1980fd8f9c6794afa8f7cb0e

Commit number `fcc46608b4ffd2cf1980fd8f9c6794afa8f7cb0e` has been covered by this audit.

UPDATE: During the audit process the team provided a minor update to accommodate deployment scripts. The updates affect the token distribution contract and have been found to be in line with the audit result and free of any known issues. The changes reviewed were included in commit number `7b356d5580dc7cc25c2efbaafb638edd5f011b60`.

UPDATE 2: The team has provided a second update in response to this audit report. The changes reviewed were included in commit number `40c3cc13eb2f39b89370289a260244a2dfba100d`

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. The following tools have been used for this:
    - Mythril - https://github.com/ConsenSys/mythril
    - Slither - https://github.com/trailofbits/slither
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

BASE CLASSES

The smart contracts build on Open Zeppelins smart contract framework release v1.12, which at the time of writing is the latest version.

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
- Token burn functionality → tokens can be burnt by the contract owner oat all time or by anyone after the contract is finalized
- Transfers of tokens to the token smart contract are blocked

Initial supply is hard-coded, and all tokens are transferred to the contract's deployer.

CROWD SALE CONTRACT

The crowd sale contract inherits from the following Open Zeppelin base contracts:

- `HasNoTokens`
- `HasNoContracts`
- `TimedCrowdsale`
- `CappedCrowdsale`
- `IndividuallyCappedCrowdsale`
- `PostDeliveryCrowdsale` (slightly modified)
- `AllowanceCrowdsale`

The contract, thus, implements a crowd sale with a fixed cap, a start and end time, a per-user account cap, and delayed delivery through a ¡n authorized wallet.

Furthermore, the crowd sale implements an additional withdrawal time limit.

DISTRIBUTION CONTRACT

The distribution contract distributes pre-sale and bonus tokens to their respective owners via time-locked escrow contracts, allows the setting of individual user caps, and, on finalization, creates the crowd sale contract.

It also acts as the token wallet for the crowd sale.

TOKEN ESCROWING CONTRACTS

A number of escrow contracts are used to implement a time locked token escrow contract which is used for pre-sale and bonus token distribution.

These escrow contracts are adaptations of Open Zeppelin's ETH escrow contracts for tokens.

SUMMARY OF FINDINGS

The contracts provided for this audit are of very good quality.

Community audited code has been reused whenever possible. A safe math library is used almost everywhere to prevent overflow and underflow issues, unless it is clearly not required. Contracts build on Open Zeppelin release v1.20.0, which at the time of audit is the latest release.

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 reasonable for this type of contract.

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 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, it is not required.

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 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 is for checking for crowd sale start and end dates, withdrawal dates and escrow time-locks. The timing tolerance in these cases is acceptable.

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

There are several places in the contracts where batch withdrawals are made by iterating over variable-sized arrays. However, a backup method is provided. See low severity issues for detail.

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 build on Open Zeppelin release v1.20.0, which at the time of audit is the latest release.

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:

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

```
   upgrade(address):    infinite

======= TokenCrowdsale.sol:TokenCrowdsale =======
Gas estimation:
construction:
   183328 + 1495400 = 1678728
external:
   fallback:    infinite
   balances(address):   598
   buyTokens(address): infinite
   cap():   526
   capReached():    887
   caps(address):   884
   closingTime():   636
   contributions(address): 708
   getUserCap(address):955
   getUserContribution(address):    1065
   hasClosed():441
   hasEnded():  1558
   openingTime():   878
   owner():882
   rate():  504
   reclaimContract(address):    infinite
   reclaimToken(address):   infinite
   remainingTokens():   infinite
   renounceOwnership():22447
   setGroupCap(address[],uint256): infinite
   setUserCap(address,uint256):21200
   token():1124
   tokenFallback(address,uint256,bytes):    857
   tokenWallet():   992
   tokensDelivered():   394
   tokensSold():    680
   transferOwnership(address): 23254
   wallet():    750
   weiRaised():548
   withdrawTime():  592
   withdrawTokens():    infinite
   withdrawTokens(address):infinite
   withdrawTokens(address[]):   infinite
internal:
   _deliverTokens(address,uint256):    infinite
   _processPurchase(address,uint256):  infinite
   _withdrawTokens(address):    infinite

======= TokenDistributor.sol:TokenDistributor =======
Gas estimation:
construction:
   infinite + 5691600 = infinite
external:
   fallback:    904
   benefactor():    860
```

```
   bonusEscrow():  486
   cap():  526
   claimUnsold(address):   infinite
   closingTime():  592
   crowdsale(): 948
   depositAndLock(address,uint256,uint256):    infinite
   depositAndVest(address,uint256,uint256,uint256,uint256):
    infinite
   depositBonus(address,uint256):  infinite
   depositPresale(address,uint256):    infinite
   depositPresale(address,uint256,uint256):    infinite
   finalize():  infinite
   getUserCap(address):infinite
   isFinalized():  954
   openingTime():  988
   owner(): 926
   presaleEscrow(): 1124
   rate():  482
   reclaimEther():  infinite
   renounceOwnership(): 22447
   setGroupCap(address[],uint256): infinite
   setTokenTimelockFactory(address):   21127
   setTokenVestingFactory(address):    21787
   setUserCap(address,uint256):infinite
   timelockFactory():  706
   token(): 1234
   transferOwnership(address): 23342
   vestingFactory():   1212
   wallet():   684
   weiRaised(): 548
   withdrawBonus(): infinite
   withdrawBonus(address): infinite
   withdrawBonus(address[]):   infinite
   withdrawPresale():  infinite
   withdrawPresale(address):   infinite
   withdrawPresale(address[]): infinite
   withdrawTime():  570
internal:
   finalization():  infinite

======= TokenTimelockEscrowImpl.sol:TokenTimelockEscrowImpl
=======
Gas estimation:
construction:
   61697 + 888800 = 950497
external:
   fallback:   332
   deposit(address,uint256):   infinite
   depositsOf(address): 735
   owner(): 552
   reclaimContract(address):   infinite
   reclaimEther():  infinite
```

```
releaseTime():  548
renounceOwnership():22161
token(): 662
transferOwnership(address): 22792
withdraw(address):  infinite
withdrawalAllowed(address): 555

======= TolarToken.sol:TolarToken =======
Gas estimation:
construction:
   infinite + 1984400 = infinite
external:
   fallback:    662
   DECIMALS():  338
   INITIAL_SUPPLY():    425
   NAME():  infinite
   SYMBOL():    infinite
   allowance(address,address): 1190
   approve(address,uint256):    22353
   balanceOf(address): 823
   burn(uint256):  infinite
   burnFrom(address,uint256):  infinite
   decimals():  618
   decreaseApproval(address,uint256):  infinite
   finalize():  22059
   increaseApproval(address,uint256):  infinite
   isFinalized():  822
   name():  infinite
   owner(): 794
   reclaimContract(address):    infinite
   reclaimEther():  infinite
   reclaimToken(address):  infinite
   renounceOwnership():22359
   symbol():    infinite
   tokenFallback(address,uint256,bytes):   747
   totalSupply():  468
   transfer(address,uint256):  infinite
   transferFrom(address,address,uint256):  infinite
   transferOwnership(address): 23122
internal:
   _burn(address,uint256): infinite

=======
crowdsale/distribution/PostDeliveryCrowdsale.sol:PostDelive
ryCrowdsale =======
Gas estimation:

======= lifecycle/Finalizable.sol:Finalizable =======
Gas estimation:
construction:
   40877 + 275800 = 316677
external:
```

```
   finalize():  21839
   isFinalized():   536
   owner(): 508
   renounceOwnership():22095
   transferOwnership(address):  22682
internal:
   finalization():  9

======= openzeppelin-
solidity/contracts/crowdsale/Crowdsale.sol:Crowdsale
=======
Gas estimation:
construction:
   61139 + 282400 = 343539
external:
   fallback:    infinite
   buyTokens(address):  infinite
   rate():  394
   token(): 530
   wallet():    486
   weiRaised(): 416
internal:
   _deliverTokens(address,uint256):    infinite
   _forwardFunds(): infinite
   _getTokenAmount(uint256):   infinite
   _postValidatePurchase(address,uint256): 13
   _preValidatePurchase(address,uint256):  89
   _processPurchase(address,uint256):  infinite
   _updatePurchasingState(address,uint256):    13

======= openzeppelin-
solidity/contracts/crowdsale/emission/AllowanceCrowdsale.so
l:AllowanceCrowdsale =======
Gas estimation:

======= openzeppelin-
solidity/contracts/crowdsale/validation/CappedCrowdsale.sol
:CappedCrowdsale =======
Gas estimation:

======= openzeppelin-
solidity/contracts/crowdsale/validation/IndividuallyCappedC
rowdsale.sol:IndividuallyCappedCrowdsale =======
Gas estimation:

======= openzeppelin-
solidity/contracts/crowdsale/validation/TimedCrowdsale.sol:
TimedCrowdsale =======
Gas estimation:

======= openzeppelin-
solidity/contracts/math/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

======= openzeppelin-
solidity/contracts/ownership/CanReclaimToken.sol:CanReclaim
Token =======
Gas estimation:
construction:
   20616 + 333200 = 353816
external:
   owner(): 486
   reclaimToken(address):  infinite
   renounceOwnership():22095
   transferOwnership(address): 22660

======= openzeppelin-
solidity/contracts/ownership/HasNoContracts.sol:HasNoContra
cts =======
Gas estimation:
construction:
   20554 + 270200 = 290754
external:
   owner(): 486
   reclaimContract(address):   infinite
   renounceOwnership():22095
   transferOwnership(address): 22660

======= openzeppelin-
solidity/contracts/ownership/HasNoEther.sol:HasNoEther
=======
Gas estimation:
construction:
   20557 + 245000 = 265557
external:
   fallback:    178
   owner(): 464
   reclaimEther():  infinite
   renounceOwnership():22073
   transferOwnership(address): 22660

======= openzeppelin-
solidity/contracts/ownership/HasNoTokens.sol:HasNoTokens
=======
Gas estimation:
construction:
   20635 + 356600 = 377235
```

```
external:
   owner(): 486
   reclaimToken(address):  infinite
   renounceOwnership():22095
   tokenFallback(address,uint256,bytes):  351
   transferOwnership(address): 22682

======= openzeppelin-
solidity/contracts/ownership/NoOwner.sol:NoOwner =======
Gas estimation:
construction:
   20794 + 487400 = 508194
external:
   fallback:   244
   owner(): 508
   reclaimContract(address):   infinite
   reclaimEther(): infinite
   reclaimToken(address):  infinite
   renounceOwnership():22117
   tokenFallback(address,uint256,bytes):  395
   transferOwnership(address): 22726

======= openzeppelin-
solidity/contracts/ownership/Ownable.sol:Ownable =======
Gas estimation:
construction:
   20504 + 192200 = 212704
external:
   owner(): 464
   renounceOwnership():22073
   transferOwnership(address): 22638
internal:
   _transferOwnership(address):22112

======= openzeppelin-
solidity/contracts/token/ERC20/BasicToken.sol:BasicToken
=======
Gas estimation:
construction:
   251 + 209200 = 209451
external:
   balanceOf(address): 581
   totalSupply():  402
   transfer(address,uint256):  infinite

======= openzeppelin-
solidity/contracts/token/ERC20/BurnableToken.sol:BurnableTo
ken =======
Gas estimation:
construction:
   349 + 310000 = 310349
external:
```

```
   balanceOf(address): 603
   burn(uint256):  infinite
   totalSupply():  402
   transfer(address,uint256):  infinite
internal:
   _burn(address,uint256): infinite

======= openzeppelin-
solidity/contracts/token/ERC20/DetailedERC20.sol:DetailedER
C20 =======
Gas estimation:

======= openzeppelin-
solidity/contracts/token/ERC20/ERC20.sol:ERC20 =======
Gas estimation:

======= openzeppelin-
solidity/contracts/token/ERC20/ERC20Basic.sol:ERC20Basic
=======
Gas estimation:

======= openzeppelin-
solidity/contracts/token/ERC20/SafeERC20.sol:SafeERC20
=======
Gas estimation:
construction:
   116 + 15200 = 15316
internal:
   safeApprove(contract ERC20,address,uint256):   infinite
   safeTransfer(contract ERC20Basic,address,uint256):
    infinite
   safeTransferFrom(contract
ERC20,address,address,uint256):infinite

======= openzeppelin-
solidity/contracts/token/ERC20/StandardBurnableToken.sol:St
andardBurnableToken =======
Gas estimation:
construction:
   1074 + 1034000 = 1035074
external:
   allowance(address,address): 882
   approve(address,uint256):   22331
   balanceOf(address): 669
   burn(uint256):  infinite
   burnFrom(address,uint256):  infinite
   decreaseApproval(address,uint256):  infinite
   increaseApproval(address,uint256):  infinite
   totalSupply():  424
   transfer(address,uint256):  infinite
   transferFrom(address,address,uint256):  infinite
```

```
======= openzeppelin-
solidity/contracts/token/ERC20/StandardToken.sol:StandardTo
ken =======
Gas estimation:
construction:
   864 + 830800 = 831664
external:
   allowance(address,address): 838
   approve(address,uint256):   22331
   balanceOf(address): 647
   decreaseApproval(address,uint256):  infinite
   increaseApproval(address,uint256):  infinite
   totalSupply():  424
   transfer(address,uint256):  infinite
   transferFrom(address,address,uint256):  infinite

=======
payment/TokenConditionalEscrow.sol:TokenConditionalEscrow
=======
Gas estimation:

======= payment/TokenEscrow.sol:TokenEscrow =======
Gas estimation:
construction:
   41391 + 695200 = 736591
external:
   deposit(address,uint256):   infinite
   depositsOf(address):647
   owner(): 508
   renounceOwnership():22117
   token(): 574
   transferOwnership(address): 22704
   withdraw(address):  infinite

======= payment/TokenTimelockEscrow.sol:TokenTimelockEscrow
=======
Gas estimation:

=======
payment/TokenTimelockFactory.sol:TokenTimelockFactory
=======
Gas estimation:

======= payment/TokenVestingFactory.sol:TokenVestingFactory
=======
Gas estimation:

======= token/ERC20/SafeStandardToken.sol:SafeStandardToken
=======
Gas estimation:
construction:
   955 + 920800 = 921755
```

```
external:
   allowance(address,address): 838
   approve(address,uint256):   22331
   balanceOf(address): 647
   decreaseApproval(address,uint256):  infinite
   increaseApproval(address,uint256):  infinite
   totalSupply():  424
   transfer(address,uint256):  infinite
   transferFrom(address,address,uint256):  infinite
```

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.

In general, gas usage is reasonable, although the distribution and factory deployment method used for the token sale is, naturally, more expensive in terms of gas then manual deployment solutions.

Note also, that the use of strings for errors in *require* and *revert* in Solidity version 0.4.24, as used in the audited contracts, currently adds a certain gas overhead, compared to previous Solidity versions. This will be optimized in future Solidity versions (https://github.com/ethereum/solidity/issues/4774).

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

POTENTIAL BATCH WITHDRAWAL BLOCK GAS ISSUES

The batch withdrawal function: of the crowd sale contract loops over a variable sized array. Should the array become very large the gas cost of the transaction may exceed the block gas limit, as it will always fail:

```
76. /**
77.  * @dev Withdraw tokens only after crowdsale ends.
78. * @param _beneficiaries List of token purchasers
79.  */
80. function withdrawTokens(address[] _beneficiaries) public {
81.     for (uint32 i = 0; i < _beneficiaries.length; i++) {
82.         _withdrawTokens(_beneficiaries[i]);
83.     }
84. }
```

This issue is marked as low severity, because a single withdrawal backup function exists. Tokens could, thus be recovered in several transactions if required.

However, in general, unbounded iterations over arrays should be avoided by splitting loops into various transactions, for example by processing 100 entries at a time.

The same issue is found in the distribution contract for bonus and pre-sale withdrawals:

```
179.        /**
180.         * @dev Withdraw accumulated balances for beneficiaries.
181.        * @param _beneficiaries List of addresses of beneficiaries
182.         */
183.        function withdrawPresale(address[] _beneficiaries) public {
184.            for (uint32 i = 0; i < _beneficiaries.length; i++) {
185.                presaleEscrow.withdraw(_beneficiaries[i]);
186.            }
187.        }
```

```
215.        /**
216.        * @dev Withdraw accumulated balances for beneficiaries.
217.        * @param _beneficiaries List of addresses of beneficiaries
```

```
218.        */
219.        function withdrawBonus(address[] _beneficiaries) public {
220.            for (uint32 i = 0; i < _beneficiaries.length; i++) {
221.                bonusEscrow.withdraw(_beneficiaries[i]);
222.            }
223.        }
```

ADDITIONAL RECOMMENDATIONS

CHANGE CROWDSALE DEPLOYMENT CONDITION

The distribution contract currently deploys the crowd sale as long as there is more than 0 Wei pending to reach the hard carp after the presale. It is recommended to raise this limit somehow, maybe to a minimum individual investment of 0.5 ETH or similar. Alternatively, a percentage of the cap might be used. Deploying a contract for very small pending investments is not very useful and, in some cases, may consume more gas than funds received through the contract.

UPDATE: The team has addressed this issue and implemented the recommendation