CrowdWiz Token Contract: Final Review

Mikhail Vladimirov and Dmitry Khovratovich

13th October, 2017

This document describes issues found in CrowdWiz contracts during code review performed by ABDK Consulting.

1. Introduction

We were asked to review a set of contracts, supplied by the customer by email on September 30th. We found several issues, one critical, and then submitted our report. The CrowdWiz authors provided us with a new revision of contracts available here. Here is the list of issues still relevant to this revision.

2. WIZ Token

In this section we describe issues related to the token contract defined in Token.sol.

2.1 EIP-20 Compliance Issues

This section lists issues of token smart contract related to EIP-20 requirements.

1. In line 158 instead `uint` should be `uint8` (according to EIP-20).

2.2 Documentation Issues

This section lists documentation issues found in the token smart contract.

1. There is no documentation for function `isContract` (line 39).
2. There is no documentation for modifiers `noReentrancy` (line 60) and `noAnyReentrancy` (line 68).
3. Semantic of the two mapping keys (line 141) is not clear.

2.3 Arithmetic Overflow Issues

This section lists issues of token smart contract related to the arithmetic overflows.

1. Correctness of the check relies on the undocumented behavior of overflow in Solidity. We recommend preventing an overflow rather than detecting it afterwards (line 6, 24).
2. There is no need for a separate function for division if "Solidity automatically throws". It might be better to prevent overflow rather than react to it (line 11).
3. In lines 108, 117 an overflow is possible. If this check is a part of a business logic, there shouldn't be any overflow as you attempt to return false. If the check is an overflow protection, it should be separated from business requirements. Also, this check will return false when `_to` is the same as `msg.sender` and number of tokens being sent plus the number of tokens belonging to `msg.sender` is greater than 2^{256}-1, while such transfer will not lead to an overflow and should be successfully performed.
4. In line 110 could be used `SafeMath` (as well as in the rest of additions and subtractions in `transfer` and `transferFrom`).
5. In line 118 there is possible overflow in case `_to` is the same as `_from` and number of tokens being sent plus number of tokens belonging to `_from` address is greater than 2^{256}-1. This line should be moved after the next line.

2.4 Unclear Behavior

This section lists issues of token smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

1. It is not clear whether `newOwner` variable in line 83 should be `public` or not.

2.5 Suboptimal Code

This section lists suboptimal code patterns found in token smart contract.

1. Value `isStarted = false` is never changing in this smart contract, but it's true for its descendants. Probably, this variable and corresponding modifier should be moved to the descendant contracts as well (line 144).
2. Modifier `onlyHolder` is not used anywhere (line 146).
3. The fallback function is not needed in case you don't want to accept Ether. If it is not defined, any Ether transfer will throw. Users would probably restrain from executing such a transaction as `Mist` or `MyEtherWallet` will issue a warning that entire gas will be spent. However, if you want the Ether transfer to fail but leap the transaction mined, you should note that the function should also be defined as `payable`. Also, the transaction will still consume some gas (line 185).
4. Function `getTotalSupply` is a duplicate of `totalSupply` getter inherited from ERC20 contract (line 189).

2.6 Moderate Issues

This section lists serious issues that may disrupt the business logic.

1. If `isContract` in line 39 returns false, the address can still be a contract. More precisely, it is possible to compute the address of a contract before deploying it as it

is deterministic. Thus, one can send Ether or token to an address with a zero codesize, and when a contract is deployed, it gets these Ether and tokens.

2.7 Double Approve Issue

The ERC-20 API is vulnerable to the double-approve attack, where an allowance recipient is able to race the spender in order to benefit from two consecutive approvals. The contract protects from it by requiring the allowance to be 0 when approve function is called. As this break backward compatibility, we recommend introducing another approval function which would have a built-in protection. Such function can be defined as `approve(address _spender, uint _oldValue, uint _newValue )` taking assumed allowance value as the second argument. If actual allowance differs from an assumed one, this method just returns false.

2.8 Other Issues

This section lists stylistic and other minor issues found in the token smart contract.

1. Functions in lines 4, 10, 17, 22 could be defined as `pure` as its value does not depend on the blockchain state.
2. Role of `Base` is not clear and it's name does not make it clear either (line 30).
3. Function `max` in line 31 and function `min` in line 32 is declared as public non-constant. It makes no sense, maybe they should be constant.
4. Condition `<=` is confusing. Its left argument is unsigned integer. Probably should be changed to `==` (line 62, 70).
5. It would make sense to allow approvals before the transfers start (line 130).

3. WIZ Crowdsale

In this section we describe issues related to the token contract defined in Crowdsale.sol.

3.1 Arithmetic Overflow Issues

This section lists issues of token smart contract related to the arithmetic overflows.

1. The logic in lines 6 and 24 relies on undocumented overflow behavior in Solidity. It would be better to check for overflow before performing multiplication.

3.2 Unclear Behavior

This section lists issues of token smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

1. Method in lines 54 and 63 may be called multiple times. It is not clear whether this is acceptable.
2. `uint transferTokens` in line 357 will round down the meaning of the mathematical expression. The remainder should be probably refunded to the sender.

3. There is no refund functionality in the contract. It's not clear whether it should be done manually (line 372).

3.3 Suboptimal Code

This section lists suboptimal code patterns found in token smart contract.

1. Function `div` in line 10 is probably not needed at all.
2. Function `add` in line 22 could be declared as `pure` as its result does not depend on the blockchain state.
3. Addresses in lines 156-158 probably should not be hardcoded, rather than be passed as constructor arguments.
4. Mapping in line 160 is initialized in the constructor and never changes, there is no need for it so it can be replaced with two address checks.
5. Line 171, `mapping (address => bool)`. It is enough to have two bits per transaction for confirmations rather than performing the whole mapping.
6. The expression in function `setState` from line 251 to 255 is overcomplicated and should be refactored. Also, switching to `EMERGENCY_STOP` state should probably be done via separate method.
7. The expressions in lines 353, 385, 450 is calculated twice.
8. Creating a new contract for every presale investor is quite expensive in terms of gas. It would make sense to self-destruct the contract inside `releaseSecond` (line 355).
9. Assignment In line 356 may result in overwriting previous `TokenTimeLock` address set with this `_to` address. It's not clear if it is supposed to be this way.
10. Addresses in lines 395-397 very similar. Would be better to mint they all once.
11. Transaction submission is possible with the destination == 0, so the check in line 138 is not reliable.
12. If bonus base would be power of two, division could be replaced with shift that is cheaper in terms of gas (line 307).

3.6 Readability Issues

This section lists cases where the code is correct, but too involved and/or difficult to verify or analyze.

1. In inequalities in line 62 and 70 the left argument is an unsigned integer, `<=` should be changed to `==`.
2. Name of the interface `ICrowdsale` is not relevant to the method declared inside (line 102).
3. Field `saleAddress` is already of type `ICrowdsale`, there is no need to cast it (line 191).
4. Interface `IToken` name is confusing because one could make a conclusion that this interface declares EIP-20 API, while it is meant to be different (line 210).
5. In line 213 `totalSupply` should probably be used instead of `getTotalSupply`.
6. There is no need to declare constructor as `public` (line 213). Constructors are always public.
7. In line 233 should be `_releaseTimeFirst < _releaseTimeSecond`.
8. Expression in line 266 would be more readable as 26e24.

9. Number in line 271 would be more readable as `1 ether`.
10. The names of parameters in line 345 are not prefixed with underscore (`_`) unlike most of the other parameter names in this file.
11. The numbers in lines 395-397 would be more readable as 4e24.

3.7 Major Flaws

This section lists major flaws found in the token smart contract.

1. `ConfirmAgent` contract looks line an extension for main `Crowdsale` contract that manages multisig confirmations of some transactions. The fact that this functionality is implemented in separate contract does not bring any value, but makes the whole system less safe because `ConfirmAgent` when being called from main `Crowdsale` contract does not have access to original msg.sender and thus have to rely on tx.origin which is easy to tamper with. Multiset logic should be merged directly into `Crowdsale` contract (line 106).
2. Default function being called with no data should fit into 2300 gas, while this functions definitely consumes more (line 400).

3.8 Moderate Issues

This section lists cases with medium priority.

1. If the function returns false, the address can still be a contract. More precisely, it is possible to compute the address of a contract before deploying it as it is deterministic. Thus one can send Ether or token to an address with a zero codesize. In this case when a contract is deployed, it gets these Ether and tokens (line 39).

3.9 Other Issues

This section lists stylistic and other minor issues found in the token smart contract.

1. Functions `mul` (line 4), `div` (line 10) and `sub` (line 17) could be declared as pure as its result does not depend on the blockchain state.
2. Synonym types `uint` and `uint256` are intermixed chaotically in the contract. Probably, just one of them should be used in every case (line 31).
3. Function `min` (line 32) is declared as public non-constant which does not make sense.
4. Address shouldn't be `public` (line 83).
5. In line 189 there should be `storage` modifier.
6. Not all branches actually return value (line 198).
7. Method should probably return bool success indicator (line 215).
8. In line 230 instead `address _token` should be `IToken`.
9. In line 152 instead `address` should be `ICrowdsale`.

4. Our Recommendations

Based on our findings, we recommend the following:

1. Fix critical and major issues which could damage the business logic of smart contact.
2. Make the token EIP-20 compliant.
3. Fix arithmetic overflow issues.
4. Check issues marked “unclear behavior” against functional requirements.
5. Refactor the code to remove suboptimal parts.
6. Simplify the code, improving its readability.
7. Fix double approve issue.
8. Fix the documentation and other (minor) issues.

5. Security provisions

We recommend the smart contract designers to claim explicitly security provisions in addition to the intended functionality of the contract. Such provisions should be non-trivial falsifiable claims, i.e. they should declare certain property of the contract for which an attack can be demonstrated if implemented incorrectly. The security provisions serve not only for the purpose of confidence of future users in the contract’s behaviour, but also as a platform for a potential bug bounty program. The contract authors are encouraged to offer various bounties binded to some provision being violated.

Here we provide a list of security provisions which are appropriate for these contracts and would be expected by ordinary users.

Name	**Description**	**Current status**
	Token	
Mint Source	Only designated minter can mint tokens	TRUE
Mint Time	Minting can occur only in the `Started` state	TRUE
Right to Start and Resume	Minter can always start or resume the state where minting is possible	TRUE
Right to Stop	Contract owner can stop the state where minting is possible	TRUE
	Crowdsale	
Price Ownership	Price and bonus values can be set by the contract owner only	TRUE

State Ownership	The crowdsale state can be changed by the contract owner only	TRUE
Direct Issuance	Issuance of tokens not backed up by Ether is granted only to predefined set of confirmOwners	FALSE (Section 3.1; confirmOwners can call a malicious contract which calls confirmTransaction).