DragonCoin Token Contract: Review

Mikhail Vladimirov and Dmitry Khovratovich

31th October, 2017

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

1. Introduction

We were asked to review a set of contracts, deployed by the customer to the Ethereum mainnet on October 2017:

- DragonBurner.
- DragonToken.
- DragonCrowdsale.
- DragonDistributions.

We got no additional documentation on the contracts so reviewed them as is.

2. Burner (a.k.a. DragonBurner)

In this section we describe issues related to the token contract defined in Burner (a.k.a. DragonBurner).sol.

2.1 Unclear Behavior (`DragonsBurned` is now Public)

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. Storage variable `DragonsBurned` is not public and there is no method for reading it, so the only way for using its value is to directly analyze the blockchain state, e.g. using `web3.eth.getStorageAt` function (line 13).

2.2 Suboptimal Code (Unclear with what you mean here)

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

1. Instead `address` should be `Dragon` (line 12).

3. Dragon (a.k.a. DragonToken)

In this section we describe issues related to the token contract defined in Dragon (a.k.a. DragonToken).sol.

3.1 Critical Flaw (Corrected this)

Method `burnFrom` does not decrease the allowance, so that it is possible for Bob to deplete the balance of Alice if Alice's allowance to Bob is at least one token (line 209) -- by repeatedly calling `burnFrom`.

3.2 Documentation Issues

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

1. Name of the function `dragonHandler` does not correlate with the name of the contract itself (line 30).
2. Meaning of the keys in mapping `(address => mapping(address => uint256))` and its submapping is unclear without documentation (line 47).

3.3 Unclear Behavior (`accountCount is now public`)

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 unclear if `_token` always the same as `msg.sender` or not (line 8).
2. Variable `accountCount` should be public (line 45).
3. There is no need to deny in case if condition `(_to == 0x0)` is abided (line 151, 187). There are many other Ehtereum addresses for which private keys are unknown.
4. Condition `if (balanceOf[_to] + _value < balanceOf[_to]) throw` may deny transfer that actually would not lead to overflow. For example in the situation then `msg.sender==_to and balanceOf[msg.sender]+_value>=2^256` (line 153, 189).

3.4 Suboptimal Code

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

1. Fields `string public name`, `string public symbol` and `string public symbol` would be more efficient as constant (lines 36-38).
2. Events `Transfer` and `Approval` do not have to be declared as they are already inherited from ERC20 (lines 51-52).
3. Event `Message` is declared but not emitted anywhere (line 53).
4. Functions `balanceOf` and `totalSupply` are automatically generated by Solidity compiler (line 117, 122).

5. Field `accountIndex` is already public, so there are two effective access methods for it (line 134).
6. `burnCheck( _to, _value )` seems to indicate two ways to burn tokens: (1) by calling burn/burnFrom method and (2)by sending tokens to the Burner smart contract. The difference between them is that the second way increases burned tokens counter in Burner smart contract while the first one does not do this. It is unclear if there should be two different ways of burning tokens and if it is reasonable to have counter once as it is easy to burn tokens without being counted (line 159).

3.5 Major Flaws (to set burner, the burner contract needs to be first deployed)

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

1. The variable `burnerSet` is used without being initialized (line 49). This is an error-prone practice.

3.6 Moderate Issues (Corrected in burnFrom method)

This section lists cases with medium priority.

1. Check `if( totalSupply - _value < 2100000000000000) throw` could be bypassed by using `burnFrom` method (line 202).

3.7 Other Issues

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

1. Most of the other contracts use Solidity version 0.4.16 whereas this one uses an older version (line 1).
2. Types `uint256` and `uint` are intermixed in the code without clear purpose (line 43).
3. Instead of `address` should be used `Burner` (line 48).
4. Variable `address` should be indexed (line 53).
5. There is no need to revoke non-consumed allowance after the call (`receiveApproval`) (line 176).

4.DragonCrowdsale

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

4.1 EIP-20 Compliance Issues (Corrected - added boolean return value)

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

1. The function `transfer` is not compatible with ERC20: as it does not return value (line 4).

4.2 Readability Issues

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

1. The value `.3333333` should be better declared as a multiple of price (line 62).

4.3 Suboptimal Code (contributions removed , excessive crowdsaleStart corrected)

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

1. The field `contributions` is never read. Perhaps there is no need to use it (line 21).
2. The condition `crowdSaleStart == false` is calculated multiple times within the same method, which could be optimized (line 62).
3. The expression `amount / price` is calculated multiple times within the same method, which could be optimized (line 74).
4. Transferring each contribution separately trough `beneficiary.transfer` is subeffective, the better way is to accumulate ether on contract's address and then transfer all at once (line 100).

4.4 Readability Issues

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

1. Contract `DragonCrowdsale` contains many long hardcoded numbers in the code which are hard to read. It would be better to turn them into named constants and then use exponential notation (line 8).
2. `package` should be local variable inside fallback function, not a state variable (line 30).

4.5 Major Flaws (2300 gas limit applies to calling another contract - this contract does not do that - tokens are received when engaging contract)

The function `payable` called with no data does not fit into 2300 gas (line 54).

5. DragonDistributions

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

5.1 EIP-20 Compliance Issues (corrected added boolean return value)

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

1. The function `transfer` is not compatible with ERC20: does not return value (line 5).

5.2 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. Contract `Dragon` was named differently in DagonCrowdsale contract (line 4).

5.3 Suboptimal Code

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

1. Instead `address` should be used `Dragon` (line 12).
2. It seems that `distributionTwo` may be performed only after `distributionOne` as well as `distributionThree` may be performed only after `distributionTwo`, so it would be more efficient to have a single mapping from `address` to the latest number of distribution performed for it, i.e. 0 (no distributions), 1 (only distribution one), 2 (distributions one and two) or 3 (all three distributions) (lines 15-17).
3. The modifier `onlyDragon` is not used (lines 23).

5.4 Readability Issues

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

1. Contract `DragonDistributions` contains long hardcoded numbers which are hard to read. It would be better to turn then into named constants and use exponential notation (line 9).

5.5 Moderate Issues (feel this is a non - issue since the waiting period is in months in order to withdraw)

This section lists cases with medium priority.

1. The assignment `clock = now` makes the distributing schedule dependant on when the transaction that deploys this contract will actually be mined (line 33).

6. Our Recommendations

Based on our findings, we recommend the following:

1. Fix the critical flaw, which puts tokens of approver at risk..
2. Fix the moderate and major flaws, which can result in large gas spending and unpredictable behavior..
3. Make the token EIP-20 compliant.
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 the documentation and other (minor) issues.