ABDK

DomusCoins Contract: Review

Mikhail Vladimirov and Dmitry Khovratovich

26th February, 2018

This document describes issues found in DomusCoin contract system during code review performed by ABDK Consulting.

1. Introduction

We were asked to review the DomusCoin Contract from the link.

We got additional documentation on the contract from the whitepaper.

2. DomusCoins

In this section we describe issues related to the contract under review. Apart from minor and moderate issues detailed below, we have identified the following crucial problems of the contracts.

First, using Oraclize for the ETH-USD exchange rate seems unnecessary. One would do that if there exists a trusted and well-formed data source, whose input can not be overridden by the contract owners so that users do not have to trust these owners for the input. Here it is not the case: the owners can easily manipulate the token price in ETH by changing the token price in USD. Moreover, the cryptocompare.com data is not guaranteed to be well-formed, and it can easily bring 0, 2^{255}, or pure garbage values for the ETH-USD rate without any liability to the ICO owners. We conclude that using Oraclize actually brings more problems than it solves, and recommend replacing this functionality with straightforward input of token price in ETH by contract owners, automated by a trusted backend if necessary.

Secondly, we observe that the sale contract is not tightly bound to the token contract. There can be many sale contracts over time, and if all of them receive tokens, it becomes difficult if not impossible to reclaim or sell them all. It is not guaranteed that how many tokens the sale contract assumes it has is equal to the real value, and such inconsistency is a source of numerous potential errors.

Thirdly, the dividend mechanism is flawed. The dividends are paid by the contract owner only by looking into the current balance of the tokenholder. It is easy to cheat this scheme by showing the same tokens many times at different accounts.

2.1 EIP-20 Compliance Issues

This section lists issues related to EIP-20 requirements.

1. Line 1115: Due to rounding errors, sum of token amounts reserved for the board will be less than `boardReservedToken` multiplied by sum of board reserved percentages. So, even if sum of board reserved percentages is 100% (what can't be guaranteed), actual number of issued tokens will be less than `tokenTotalSupply`. This violates EIP-20 standard.

2.2 Documentation Issues

This section lists documentation issues found in the smart contract.

1. Line 1025: `Math` library does not provide any new math operations but rather makes existing operations safer. Name `SafeMath` should be used instead of `Math`.
2. Line 1092: the mapping `allowed` has two keys and their meaning is unclear without comments.
3. Line 1102: the comment is incorrect. The code allows arbitrary percentage, not just 90%.
4. Line 1103, 1115: if `_publicReservedPersentage` and `boardReservedPersentage` are supposed to be not greater than 10^4, this should be documented and an appropriate `require()` check inserted.
5. Line 1224: check `(proposedOwner == address(0))` assumes that `proposedOwner` can not be set to 0. Probably it would be better to add the comment describing this information.
6. Line 1278,1288: there should be `addTotalToken` instead of `setTotalToken`.
7. Line 1499, 1502: `_newAddress`, `_beneficiary` parameter should probably be indexed.
8. Line 1562: looks like the valid price is actually between $1 and $1000 inclusive.
9. Line 1581: the name of function `setTotalToken` is confusing. Also, the method adds tokens rather than sets the number.
10. Line 1581: `_token` is confusing, it would be better to rename it to `amount`.
11. Line 1657: the function name `startStopUpdateTokenPerEther` is confusing.
12. Line 1737: the case then $999.90 will render as $999.9 not taken into account.
13. Line 1757: the input parameter of `FlexibleTokenSale` is `tokenPerEther`.

2.3 Readability Issues

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

1. Line 1089, 1090, 1092: the fields `publicReservedToken`, `tokenConversionFactor` and `allowed` use default access. The code would be more readable if access would be specified explicitly.
2. Line 1090: there’s no need to use brackets in this line.

3. Line 1154, 1164, 1165 : in this, library `Math` is used as a part of business logic to ensure that sender has enough tokens to send. According to Solidity docs, `require` should be used for such kind of checks where `Math` is using `assert` instead. Also, hiding such important business-logic into generic method makes code harder to read.
4. Line 1253, 1256: the fields `publicReservedAddress` and `boardReservedAccount` uses default access. The code would be more readable if the access were specified explicitly.
5. Line 1285: methods that do not return value look like they modify state, whereas `validateTransfer` is pure and its sole purpose is to throw exceptions if some condition is not satisfied. This makes code less readable. Probably the method should return boolean or be converted to a modifier.
6. Line 1309: the rest of this method is effective in `else` branch of this statement. The code would be more readable if `else` condition were added.
7. Line 1518: to improve readability there should be `5 ether` instead of `5 * 10**18.`
8. Line 1582: splitting `(msg.sender == address(token) && _token > 0)` into two separate `require` calls would make code more readable.

2.4 Unclear Behavior

This section lists issues of the 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. Line 1223: the purpose of `cancelOwnershipTransfer()` method is unclear. In fact, an initiated owner transfer can always be reverted by calling `initiateOwnershipTransfer` one more time.
2. Line 1225: the meaning of the return value is unclear. If it is a success indicator, then a situation when one tries to cancel ownership transfer that wasn't actually initiated is probably good reason to return false.
3. Line 1296, 1352: the purpose of check `(_to != address(0)` and `address(_saleToken) != address(this)` is unclear. Is it a protection from multiple use?
4. Line 1299: `address(saleToken) != address(0)` check looks unnecessary: if `saleToken` address equals zero, a message sender can’t be in the line below.
5. Line 1319: it is unclear from the code that all board members eventually get their tokens. If this is a requirement, the logic should be simplified to assert that.
6. Line 1586: event in `setTotalToken` is the same event as emitted in `setMinToken`.
7. Line 1664: the contract owner can arbitrarily set the token price in USD. Using external oracle to get the ETH-USD conversion rate is redundant: just allow the contract owner to set the price in ETH as well, and you would not need Oraclize at all, just call the price update method as frequently as you want (it's already done). Also, the Oraclize smart contract might have the ETH-USD rate ready as is, as it calculates the ETH price for requests given the fixed price in USD: http://docs.oraclize.it/#pricing-advanced-datasources-call-fee.

8. Line 1672: `updateTokenPerEther` looks like a recursive call. It could be incorrect.

2.5 Arithmetic Overflow Issues

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

1. Line 1031, 1048: correctness of checks `assert (r >= a)` and `assert(a == 0 || r / a == b)` relies on the undocumented behavior of overflow in Solidity. We recommend preventing an overflow rather than detecting it afterwards.
2. Line 1740: two overflows are possible in these places:
   - `result1 * 10;`
   - `+ (c - 48).`

2.6 Suboptimal Code

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

1. Line 1054: the function `div` seems to be a useless wrap around the standard integer division operation.
2. Line 1095: perhaps there is no need to have values `string _name, string _symbol, uint8 _decimals` configured at deployment time. It might be better to hardcode them to make contract cheaper to deploy and use.
3. Line 1103, 1115: `uint256` conversion actually does nothing.
4. Line 1200: check in this line is redundant.
5. Line 1211, 1212: checks `(_proposedOwner != address(0)` and `(_proposedOwner != address(this)` are unclear. Are they to prevent multiple calls of the method?
6. Line 1306: check `if (allowed == 0)` doesn't cover the case when board member has zero percentage allocated.
7. Line 1345: function `currentTime()` is not used in this smart contract.
8. Line 1385: percentage denominator 10000 should be made named constant rather than hardcoded value. Otherwise, percentages are hard to understand.
9. Line 1477: `TokensReclaimed` event looks redundant because any token transfer from the address of this token contract is a reclaim.
10. Line 1479: `totalToken` seems to duplicate the token balance of the sale contract, which is error-prone as these two values are not guaranteed to be equal.
11. Line 1531: the expression `10**(uint256(18).sub(_token.decimals()).add(4).add(2))` can be simplified.
12. Line 1532, 1651: `assert` should be used instead of `require`.
13. Line 1537, 1557,1567, 1575: `return` control is redundant.
14. Line 1650: `token.balanceOf(address(this))` may be executed even if the smart contract is not yet initialized and token address is not set.
15. Line 1654: there should be a check that `buyTokensInternal` has not failed so the transaction can be reverted.
16. Line 1735: `if(c == 46)` is redundant because all characters other than digits are skipped anyway.

2.7 Major Flaws

This section lists major flaws that require special attention.

1. Line 1334: board members can't retrieve tokens if `publicReservedToken` equals zero.
2. Line 1350: permitting of `setICOAddress` as calling method between transfers may disrupt the entire logic in `getBoardMemberAllowedToken` and similar methods.
3. Line 1365: the comment is incorrect. The consistency of `totalSupply` with actual token allocation is never actually asserted. Moreover, the smart contract written like `totalSupply` will differ from the actual number of tokens in circulation, due to rounding errors in the constructor of the base class and due to the fact that sum of board percentages is not guaranteed to be equal to 100%.
4. Line 1426: 999 means 9.99%, not 99.9%.
5. Line 1427: it seems that dividend percentage may be changed after it was set, also it may be changed at the moment when some users already got their dividend and some didn't get dividends yet. Taking into account that dividend payouts and dividend percentage changes do not log anything, this could cause the total mess.
6. Lines 1548-1551: checks do not bring extra protection as there can be numerous other bad addresses, and all of them can be corrected by another call.
7. Line 1551: `isOwner(_walletAddress) == false` should be replaced by `!isOwner(_walletAddress)`.
8. Line 1666: `oraclize_query` depends on conversion rate, encoding and etc. It looks dangerous
9. Line 1672, 1743: new value `tokenPerEther` is assigned without any sanity checks. This could be dangerous in case cryptocompare.com will experience some problems.
10. Line 1719: if `totalToken` value is inconsistent with the token balance (which happens if someone sends the tokens to the sale contract when the latter is not yet designated as such) the method will not work and tokens will be lost forever.

2.8 Moderate Flaws

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

1. Line 1095: the number of elements of the arrays `address[] boardReserved,uint256[] boardReservedPersentage` implicitly limited by block gas limit. This limitation could be removed by allowing adding more board reserves after the contract is deployed.
2. Line 1113, 1268: loops exhausts entire gas (for arrays of the certain size).
3. Line 1224, 1259, 1526, 1584: variables `proposedOwner, saleToken, token` and `totalToken` are used without being initialized.
4. Line 1314: method `getBoardMemberAllowedToken` is stateless, so the caller can transfer allowed the number of tokens arbitrarily many times. Thus, the entire check is reduced to boolean `allowed>0`.

5. Line 1334, 1337: due to rounding errors, the resulting value for `publicReservedSoldPersentage`, `remainToken` will be lower than the correct value.
6. Line 1337: the function `publicReservedSoldPersentage` is already rounded down. The resulting value of `remainToken` will be lower than correct value.
7. Line 1338: because `remainToken` value is lower than the correct value, resulting value for `allowedToken` will be higher than the correct value. This allowing board member to transfer more tokens that they are supposed to be able to transfer.
8. Line 1415: the name of function `claimDividendTokens` is confusing. The method can not be called by a token holder, only by the owner. Moreover, the transfer is not recorded anywhere, so the method does the same as a regular token transfer from the token contract balance to an address.
9. Line 1418: token holder may get dividends for the same tokens second time by transferring tokens from the address that already got dividends to address that is about to get dividends.
10. Line 1683: rounding error occurred in `div` will be multiplied by `msg.value` and then by 10000 which may make it significant.

2.9 Other Issues

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

1. Line 1261: the parameter `burner` probably should be indexed.
2. Line 1406: the function `dividendPersentage` should be initialized to some default value.
3. Line 1631: the function `payable` consumes more than 2300 gas, which violates Solidity guidelines.

3. Our Recommendations

Based on our findings, we recommend the following:

1. Further protect or remove the Oraclize integration.
2. Fix the dividend scheme.
3. Ensure that only one sale contract ever exists.
4. Fix the moderate flaws, which restrict functionality due to gas limits.
5. Make the token EIP-20 compliant by fixing total supply issues.
6. Fix arithmetic overflow issues.
7. Check issues marked “unclear behavior” against functional requirements.
8. Refactor the code to remove suboptimal parts.
9. Simplify the code, improving its readability.
10. Fix the documentation and other (minor) issues.