eSports Token Contract: Review

Mikhail Vladimirov and Dmitry Khovratovich

24th October, 2017

This document describes issues found in eSportsCoin 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 12 October 2017:

- ESports Freezing Storage.
- ESportsConstants.
- ESportsToken.
- ESportsMainCrowdsale.
- Zeppelin/ESportsCrowdsale.
- ESportsBonusProvider.

We got additional documentation on the contracts from the eSports Whitepaper and Deploy Instruction.

2. ESportsFreezingStorage

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

2.1 Suboptimal Code

Because of the instructions in lines 24 and 32 (`return 0`) it is impossible to distinguish an unsuccessful attempt to release non-zero value from the successful release of zero value. The commented code lines would work better and would allow a more efficient discard of used bonus contracts.

2.2 Critical Flaw

The method `release` is available to any address (line 22) so after tokens are unlocked, anyone can withdraw them to his own address, not only the intended beneficiary. It is recommended to make it available only to token owner or the crowdsale contract.

2.3 Other Issues

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

1. Instead of freezing the storage it might be better to make the contract self-destruct itself after frozen tokens at released as it does not have any utility since then (line 7).
2. In Ethereum `uint256` is usually used for storing timestamps (line 9).

3. ESportsConstants

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

3.1 Suboptimal Code

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

1. Constant in line 5 looks redundant and should probably be removed.

3.2 Readability Issues

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

1. It would to be better to start contract name with the capital letter. Also, the name is confusing as `using` is a keyword meaning use of external library (line 3).

3.3 Other Issues

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

1. Most of other contracts use Solidity version 0.4.16 whereas this one uses an older version(line 1).
2. Probably, `TOKEN_DECIMALS` could be used instead of hardcoded value at line 5.

4.ESportsToken.sol

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

4.1 EIP-20 Compliance Issues

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

1. According to EIP-20 instead `bytes32` should be `string` (line 27).

4.2 Documentation Issues

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

1. In line 13 instead `true` should be `false`.

4.3 Arithmetic Overflow Issues

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

1. In line 85 arithmetic overflow is possible.We recommend to use `SafeMath` here.

4.4 Suboptimal Code

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

1. Creating new contract for time-locked tokens each time (line 70) is gas-consuming. It is possible to store all the time-locked tokens in the same contract, probably exactly this one, and then use state variables to keep a track on token owners and release times. The current code does not kill contracts from which tokens have been requested; we'd recommend that to clean the blockchain. See also Section 2.2.
2. The loop starting in line 84 may consume arbitrary large amount of gas, so it cannot be safely used from other contracts. Probably it would be better to store a total frozen amount per beneficiary and then update this stored value when necessary, rather than calculating every time.
3. Expression `frozenFunds[_beneficiary]` is calculated twice per loop iteration. While Solidity could be smart enough to optimize this, it would be better to calculate it once before the loop and then store it into local variable (line 85).
4. Accessing `msg.sender` is cheaper than accessing local variable (or at least not more expensive), so line 113 is redundant.

4.5 Readability Issues

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

1. The `crowdsaleFinished` naming of the function is confusing. It does not look like the name of function that changes contract's state, but rather like name of a modifier or constant function. Also, the function does not end the crowdsale. Instead, it disables the pause, which may happen at any point of time (line 35).
2. `SafeMath` is implicitly used In line 114 to check that `msg.sender` has enough tokens to burn. It would make code more readable if this check will be made explicitly.

4.6 Moderate Flaws

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

1. Function `x < frozenStorages.length` may consume arbitrary amount of gas, resulting in a situation when if one beneficiary has too many tranches of frozen funds, these funds may effectively become locked (line 98).

4.7 Other Issues

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

1. It might be reasonable to log an event here with `_to` and `_amount` as parameters so that the token holders can easily track their frozen tokens and release dates (line 76).

5. ESportsMainCrowdsale

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

5.1 Documentation Issues

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

1. Code would become more readable if there was a constant for the total amount of tokens and all the percentages applied to it (line 12).
2. The whitepaper is set as `4 000 000` euros as the lower cap whereas it is mentioned 2 000 000 wei (?) in the comment (line 51).

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. Addresses in lines 20-28 are suspicious. Several of its addresses are active in the Kovan testnet, while others are not used in the code or are not active in the mainnet, and thus are unlikely to become multisig contracts.

5.3 Suboptimal Code

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

1. Function `onlyOwner` in line 113 is redundant as the function is internal and could be called from init, which is available to the owner only.

5.4 Readability Issues

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

1. The `internal` non-public function in line 68 is intermixed with the public one which makes it harder for the reader to realize which one is public API and which one is not.

5.5 Logic Discrepancy

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

1. Maybe `_hardCapTokens` should be listed as hard-coded constant (line 41) since in the whitepaper it is fixed.

5.6 Other Issues

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

1. In Ethereum timestamps `uint256` are typically used, while in `ESportsFreezingStorage` smart contract `uint64` is used, resulting in the different types mentioned for the same thing which is confusing and error-prone (line 38).
2. Function `init()` consumes a lot of gas, better check if it fits the block gas limit (line 81).
3. Expression in line 86 (`ESportsToken(token)`) is repeated several times. Probably it should be better to store to local variable.

6. EsportsCrowdsale

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

6.1 Documentation Issues

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

1. In description in lines 7-8 explicitly should be added that this code differs from Zeppelin's codebase, as it is not obvious from the repository structure.

6.2 Arithmetic Overflow Issues

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

1. In line 121 `SafeMath` should be used.

6.3 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. The `validPurchase` method verifies if at least one token can be bought whereas it might make sense to check for the full amount. If so `_actualRate` should probably be used `_amountWei.mul(_actualRate)` (line 161).
2. In case if `getRate` is overriden it may return value less than value stored in `rate` state variable line 171. Probably, `getRate()` should be used instead of `rate`.

6.4 Suboptimal Code

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

1. If `buyTokens` method were `public` rather than `internal`, the tokens could be purchased from a multisig wallet (line 90).

2. Usually Solidity will not calculate right argument of `&&` is case left parameter is false to save gas, but here all three arguments are precalculated making code less efficient (line 163).

6.5 Other Issues

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

1. Function `payable` does not fit into 2300 gas when called with no data, which violates Solidity guidelines (line 85).
2. There is no provision for case if `msg.sender` is a contract whose fallback function is not payable or does not fit into 2300 gas (line 139).
3. Function `postBuyTokens` in line 152 makes the whole contract to be abstract. Probably should have default no-op implementation.
4. Looks like `msg.value` is used as hidden parameter modifying behavior of `buyTokens` functions. Probably it would be better to make this parameter explicit such as bool `_isBTCPurchase` (line 105).

7. ESportsBonusProvider

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

7.1 Documentation Issues

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

1. The meaning `I` in line 9 not clear. Perhaps `I` mean “Interface”, but this contract does not look like interface. It has fields and implemented methods, it also inherits contract “Ownable” that is not an interface.
2. Meaning of return value `uint` is unclear without a documentation (line 25,27).
3. Semantic of `releaseBonus` in line 27 is unclear without a documentation.
4. Function `constant returns` in line 99 is “public” but it’s not specified explicitly, in contrast to the other public functions in this contract.

7.2 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. Perhaps condition in line 64 should be mentioned as `&& now >= _startTime.`
2. The token counting operation performs division and then multiplication. So it might end up in accumulating more and more errors. Probably it would be better to add `_amountTokens` values as is and divide by 10 inside `releaseBonus` method (line 81).

7.3 Readability Issues

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

1. In line 47 instead `7 * 1 days` would be more readable `7 days`.
2. `uint` in lines 60, 76 could be changed to `uint bonus` to simplify code.

7.4 Other Issues

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

1. According to Solidity documentation, `require` is for validating input, state and output which is not the case here, so probably `revert` would be more appropriate (line 30, 94).
2. `remainBonusTokens` is zero. Calling `transfer` here is unclear (line 35).

8. Our Recommendations

Based on our findings, we recommend the following:

1. Fix critical flaw which could damage the business logic of smart contact.
2. Fix the moderate flaws, which can be result in arbitrary amount of gas.
3. Make the token EIP-20 compliant.
4. Fix arithmetic overflow issues.
5. Check issues marked “unclear behavior” against functional requirements.
6. Refactor the code to remove suboptimal parts.
7. Simplify the code, improving its readability.
8. Restore the logic (section Logic Discrepancy).
9. Fix the documentation and other (minor) issues.