Go For It Token Smart Contract Code Review

G. Baecker

July 2019

CONTENTS

1 Introduction

1.1 About

In July 2019 we—the Tecneos UG—were asked to do a code review of the Ethereum smart contract `GoForItToken` written in the Solidity programming language.

While doing the code review we were in close contact to the programming team. This allowed us to point out possible issues and make suggestions for code changes which were either incorporated directly or fixed by other means.

This document summarizes our final findings.

1.2 Code Version

The smart contract source code was fetched from:

Git Repo URL	http://46.101.229.186:3000/sicos/goforit2.git
Commit Hash	63f0ad8eb0a119b3e50ceb2e690ff4da116a7e1e
Commit Date	2019-07-11 18:42:07

1.3 Scope

The code review was solely about the smart contract `GoForItToken.sol`.

The code base depends heavily on the OpenZeppelin Solidity Framework which was not subject to the code review.

1.4 Disclaimer

This report is not a legally binding document and it doesn't guarantee anything. The reader should be aware that it's in the nature of things, that revealing/eliminating all potential/real problems in a non-trivial software project is impossible.

Nevertheless, this report is a result of careful work done for the best.

This is just a discussion document.

2 Code Review

2.1 Summary

The smart contract `GoForItToken` has a simple code structure as it is completely based on well-known *OpenZeppelin Solidity framework's*

- `ERC20Burnable`
- `TokenVesting`

We consider this a good practice because of OpenZeppelin's very good reputation within the *Ethereum* community for its secure and reliable smart contracts code base which regularly undergoes security audits.

The custom code extensions in `GoForItToken` are straight-forward.

No issues were found.

2.2 Details

2.2.1 Major Issues

None found

2.2.2 Minor Issues

None found

2.2.3 Notes

The following is merely related to code style and does not affect the integrity of the `GoForItToken` smart contract.

Usage of unit `ether` in non-Ether context

In line 9 the total token supply is defined as:

```
9       uint constant TOTALTOKENSUPPLY = 12500000000 ether;
```

This is numerically fine since the token is defined to have 18 decimals. But it is not quite correct regarding the unit. Token amounts are not denominated in Ether.

Proposal

This is a private constant. So the following will compile to a literal value, i.e. calculated by the compiler, without pretending to be some Ether amount:

```
    uint constant TOTALTOKENSUPPLY = 12500000000 * 10**18;
```

Naming of Constant

The Solidity Style Guide recommends to name constants "with all capital letters with underscores separating words".

Proposal

Consider renaming TOTALTOKENSUPPLY to TOKEN_TOTAL_SUPPLY for increased code readability.

Multiple Array Lookups for Same Value

The constructor accesses four or two times—depending on the code branch taken—the same array value in each iteration step:

```
require(vestingContracts[beneficiaries[i]]== address(0), "only 1 contract per address");
    if(vestingDays>0) {
    TokenVesting vestingContract =new TokenVesting(beneficiaries[i],now,vestingDays* 1 days,ves
    _mint(address(vestingContract),amounts[i]);
    vestingContracts[beneficiaries[i]]=address(vestingContract);
    emit TokenVested(beneficiaries[i],address(vestingContract),amounts[i]);
    }
    else {
    _mint(beneficiaries[i],amounts[i]);
    }
```

Proposal

To increase code readability and to reduce deployment costs consider using a local variable:

```
address beneficiary = beneficiaries[i];
require(vestingContracts[beneficiary] == address(0), "only 1 contract per address");
if(vestingDays > 0) {
    TokenVesting vestingContract = new TokenVesting(beneficiary, now, vestingDays * 1 days, ves
    _mint(address(vestingContract), amounts[i]);
    vestingContracts[beneficiary] = address(vestingContract);
    emit TokenVested(beneficiary, address(vestingContract), amounts[i]);
}
else {
    _mint(beneficiary, amounts[i]);
}
```

Multiple Computation of Same Value

The constructor calculates the vesting duration two times.

Proposal

Consider doing this calculation by changing line 30 to:

```
30  uint vestingDuration = vestingInDays[i] * 1 days;
```

and using this value in:

```
32  if (vestingDuration > 0) {
33      TokenVesting vestingContract = new TokenVesting(beneficiaries[i],
34                                                      now,
35                                                      vestingDuration,
36                                                      vestingDuration,
37                                                      false);
```

Inconsistent Behavior in Edge Case

While it is assumed that all beneficiaries given to the constructor are distinct, it is possible for the `beneficiaries` parameter to contain the same address several times if not more than one related vesting duration is non-zero.

The following code snippet shows two constructor calls which seem to lead to similar behaving contracts. But while the first one works, the second will fail due to the order of values in `vestingInDays` argument—thus the inconsistency:

```
address beneficiary = 0x...;
GoForItToken token1 = new GoForItToken([beneficiary, beneficiary], [0, 7], ...);
GoForItToken token2 = new GoForItToken([beneficiary, beneficiary], [7, 0], ...);
```

Proposal

This is a theoretical edge case; irrelevant in practice. Leave it as it is.

Code Style, Indentations, Whitespaces

The source code doesn't follow the Solidity Style Guide suggestions regarding code layout, especially indentation, whitespaces, maximum line length.

Proposal

Reformat the code. Make use of a linter (e.g. Solhint, Solium, ...).

Code Comments

The source code is not documented.

Even if the behavior is mostly obvious to adept readers, comments are useful for

- communicating the programmer’s intentions, thus these can be easily compared to the actual behavior (and allow to reveal possible mistakes)
- automated generation of documentation if given as NatSpec comments

□