Shping
Smart Contract Version 2: Review

Mikhail Vladimirov and Dmitry Khovratovich

19th January, 2018

This document describes the issues, which were found in the Shping smart contract during the code review performed by ABDK Consulting.

1. Introduction

We were asked to review a set of contracts in the contracts folder at the commit `967dd7`.

2. ShpingCoin

In this section we describe issues related to the smart contract defined in the ShpingCoin.sol.

2.1 Critical Issues

This section lists a critical flaw, which was found in the smart contract.

1. Line 45: This check looks very odd. Should it be `newOperator != operator`? If `newOperator` is the same as `operator`, this will effectively wipe out operator's balance without properly updating `_totalSupply`, thus making contract state inconsistent.

2.2 Documentation and Readability Issues

This section lists documentation issues, which were found in the smart contract, and the cases where the code is correct, but too involved and/or complicated to verify or analyze.

1. Line 12 : Equivalent types “uint” and ”uint256” are intermixed in the code. Code will be more readable if the same notation will be used everywhere.
2. :Line 14: Usually, underscore prefix is used for parameter names, not for state variables.
3. Line 16,19: This mapping has two keys whose meaning is not obvious without documentation comment.

4. Line 18, 65: `string` seems to be fixed-size hash of something, so bytes32 or uint256 or some other fixed-size type could be more effective here.
5. Line 18: State variables are usually named in camelCase, not c_style.
6. Line 27: Operator precedence is not obvious here, brackets will make code more readable.
7. Line 119: Code will be simpler if this state variable will be renamed to totalSupply and declared as public, and method totalSupply() will be removed.

2.3 Arithmetic Overflow Issues

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

1. Line 44: This overflow check looks like part of business logic which makes code harder to read. Consider using SafeMath or at least change “require” to “assert”. Also this check denies to change operator to itself in case operator's balance is greater then 2^255-1.
2. Line 68, 89: Underflow is possible here.
3. Line 90, 103, 128, 139: Overflow is possible here.
4. Line 129, 141: Another overflow check that looks like business logic. Also this check will fail if value is zero (directly violates EIP-20). Also, this check will fail if msg.sender and “to” are the same and current token balance of msg.sender plus value being transferred is greater than 2^256-1.

2.5 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 42: This method clears the operator's balance but does not check his budget. Thus operator can get rid of his tokens with his budget still being non-zero.
2. Line 43, 86, 96: Why to disallow assigning address to 0?
3. Line 81: Event `Reject` will be logged even if there were no campaign with given parameters.
4. Line 81: True will be returned even if there were no campaign with given parameters.

2.6 Suboptimal Code

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

1. Line 5: Structure of one fields looks odd, could be replaced with bool type.
2. Line 113: Is it really necessary to index by budget field?
3. The same: This looks like one of the keys used to identify campaign, so probably this parameter should be indexed.

2.7 Other Issues

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

1. Line 45: This does not emit Transfer event making is harder to track token transfers.
2. Line 85: Method `setBudget` should probably log an event.

3. Our Recommendations

Based on our findings, we recommend the following:

1. Fix the most important flaw described in section 2.1.
2. Fix arithmetic overflow issues.
3. Check the issues marked as “unclear behavior” against functional requirements.
4. Refactor the code to remove suboptimal parts.
5. Fix the documentation issues.