Pillar Token Code Review

July 14, 2017

Prepared By: Kshitish Balhotra

kshitish@dltlabs.io

Independent Reviewers
Umesh Kushwaha,
Bhavish Balhotra

dltlabs.io

Table of Contents

I. **Introduction** 2

II. **Overview** 4

III. General Findings 4

1 Introduction

DLT Labs conducted a review of the smart contracts that make up the Pillar Token Sale service. The findings of the review are presented in this document.

This review was performed under a contracted fixed rate and performed by independent reviewers.

Update: The Pillar Token Development team has informed us that they have deployed the fixes to the issues observed in our reviews and details of which are stated below. DLT Labs' independent reviewers have verified the fixes for the issues stated below and they can be marked as fixed from a review perspective.

1.1 Review Goals and Focus

1.1.1 *Sound Architecture*

This review includes both objective findings from the contract code as well as subjective assessments of the overall architecture and design choices. Given the subjective nature of certain findings it will be up to the Pillar Token development team to determine the appropriate response to each issue.

1.1.2 *Smart Contract Best Practices*

This review will evaluate whether the codebase follows the current established best practices for smart contract development.

1.1.3 *Code Correctness*

This review will evaluate whether the code does what it is intended to do.

1.1.4 *Code Quality*

This review will evaluate whether the code has been written in a way that ensures readability and maintainability.

1.1.5 *Security*

This review will look for exploitable security vulnerabilities.

1.2 Terminology

This review uses the following terminology.

1.2.1 *Code Coverage Terms*

Measurement of the testing code coverage.

1.2.1.1 Untested

No tests.

1.2.1.2 Low

The tests do not cover some set of non-trivial functionality.

1.2.1.3 Good

The tests cover all major functionality.

1.2.1.4 Excellent

The tests cover all code paths.

1.2.2 *Severity Terms*

Measurement of magnitude of an issue.

1.2.2.1 Minor

Minor issues are generally subjective in nature, or potentially deal with topics like *"best practices"* or *"readability"*. Minor issues in general will not indicate an actual problem or bug in code.

The maintainers should use their own judgement as to whether addressing these issues improves the codebase.

1.2.2.2 Medium

Medium issues are generally objective in nature. Most medium level issues will not represent an actively exploitable bugs or security problem, but rather an issue that is *likely* to lead to a future error or security issue.

In most cases a medium issue should be addressed unless there is a clear reason not to.

1.2.2.3 Major

Major issues will be things like bugs or security vulnerabilities. These issues may not be directly exploitable such as requiring a specific condition to arise in order to be exploited.

Left unaddressed these issues are highly likely to cause problems with the operation of the contract or lead to a situation which allows the system to be exploited in some way.

1.2.2.4 Critical

Critical issues are directly exploitable bugs or security vulnerabilities.

Left unaddressed these issues are highly likely or guaranteed to cause major problems or potentially a full failure in the operations of the contract.

2 Overview

2.1 Source Code

The source code under review can be found in the public github repository https://github.com/twentythirty/PillarToken/

2.2 Contracts

The tests were primarily limited to test files related to the main sale namely PillarToken.sol and unSoldAllocation.sol

2.2.1 *Zeppelin Base Contracts*

The contracts make use of a small number of the re-usable contracts provided by Zeppelin. These contracts were not covered by this review.

3 General Findings

This section contains detailed issues and analysis.

3.1 General Thoughts

The first review identified multiple issues that require patching prior to the contracts being deployed and used. The changes necessary to fix these issues are likely to change a sufficiently large amount of code to warrant an additional follow up review. This was reported in our meeting held on 5th July 2017. The following issue numbers were identified in the first review: Critical: 3.4.1-3.4.2, Major: 3.3.1-3.3.3, Minor: 3.2.1-3.2.4

A second review was conducted to further review the previously identified issues and also see if any new issues were created due to code update. The issues have been detailed in relevant sections below.

A third review was conducted at the request of the Pillar Project team to further review the previously identified issues and also see if any new issues were created due to code update. The issues have been detailed in relevant sections below. All issues stated below have been fixed and reviewed.

3.1.1 *Code Quality*

The code reviewed in this review is generally of good quality. We do feel that there is scope to improve code verbosity by adopting the latest standards set for token contracts and also improve the comments provided.

The contracts themselves could have been architected and organized as per design standards. Commonly used logic for token sale has been encapsulated in internal utility functions. The Token contract is ERC-20 compatible and uses standard Zeppelin libraries. The Zero address attack have also been handled appropriately.

3.2 Minor Issues

3.2.1 *Remove isFundable modifier*

In line 134, isFundable modifier is not required as function fundingActive() is required to return current funding status whether funding is active or not.

3.2.2 *totalUsedTokens condition implementation needs to be fixed*

In line number 135, condition totalUsedTokens > totalAvailableForSale will never be met so funding status will show as active only even though all available token has been sold.

3.2.3 *No specific function exists to start token sale*

As per current implementation, Sale for PillarToken will be started as soon as smart contract is published. It is advisable to add specific function for starting token sale so that owner can start sale at specified time avoiding any last moment deployment issues.

3.2.4 *allocateTokens purpose needs to be defined*

The purpose of function allocateTokens has not been clearly defined due to which it is difficult to assess its usage. We also need to clearly identify as to why are we subtracting token assigned from totalUsedTokens.

3.2.5 *Refund function calculation formula needs to be fixed*

In function refund() at line number 161, ethValue for refund is calculated as below: uint ethValue = plrValue.mul(tokenPrice); This should be modified to uint ethValue = plrValue.mul(tokenPrice).div(1e18); because tokenPrice is for per token but plrValue is holding token amount multiplied with 1e18 (as per line number 103).

3.2.6 *Implement unPauseTokenSale function to unpause sale*

We need to have unPauseTokenSale() function in contract that will set fundingMode to true so that paused sale can be resumed. The current implementation does not have any function that can be used to resume paused token sale leading to a deadlock state. This issue was reported in first review, the incorporated fix provided after second review has not taken care of the scenario mentioned here.

3.2.7 *Token Sale finalization issue when we hit max token limit*

As per the current implementation, Token sale cannot be finalized in case totalUnSold =0. This causes issues when we hit the limit for the tokens to be sold hence leaving the token sale open

even though there are no more tokens to be sold. We suggest removing condition if (_tokens <= 0) throw; from line number 27 of UnsoldAllocation.sol so that the sale can be finalized.

3.3 Major Issues

3.3.1 *No Implementation of unPause function*

Token sale cannot be un-Paused, if once it is paused as there is no implementation for un-pause.

3.3.2 *totalSupply function implementation logic*

In fallback function, as per line number 90 & 99, token sale is allowed till totalSupply for direct transfers of ether to contract. In this scenario, it may lead to over sale of token because all totalSupply is not available for sale.

3.3.3 *allocateForRefund function needs to accept ether*

In line number 194, function allocateForRefund must be payable to accept ether. Refunds will not be successful until smart contract is funded with ether using function allocateForRefund(). This is not automated process so business to take prompt action in case ICO is not successful to avoid any chaos.

3.3.4 *Set salePeriod, fundingStartBlock and fundingStopBlock in startTokenSale*

In order to show accurate data, salePeriod, fundingStartBlock and fundingStopBlock should be set in function startTokenSale() not in constructor as these parameters should be set while starting sale not at point of deployment. This is necessitated by the fact that the salePeriod gets calculated based on current block timestamp, so it will show wrong data if set in constructor.

3.3.5 *Update the condition for the finalize function*

In line number 134, below condition for function finalize() needs to be corrected as in the current implementation, sale can be finalized when block.number > fundingStopBlock even though minTokensForSale are not sold.

```
if ((block.number <= fundingStopBlock && totalUsedTokens < minTokensForSale)) throw;
```

3.3.6 *Set fundingStatus as a constant keyword*

In line number 206, function fundingStatus() should include 'constant' keyword in signature to get the value from function, as this function is not changing any state variable. This is needed in order to ensure that we are able to query a response for the status while the sale is on.

3.4 Critical Issues

3.4.1 *Decimal Placement*

As PillarToken supports 18 places of decimals, it means all transactions will happen in minimum possible unit and also balances of any address will be stored in minimum possible unit not in PillarToken. And all other wallets will understand as same way. So as per current implementation 1000 PillarToken received from PillarToken smart contract will display balance as 0.000000000000001000 PillarToken in other wallets.

3.4.2 *totalUnSold exception*

In line number 168, totalUnSold will always be calculated as 0 because finalize function will be executed only when either block.number > fundingStopBlock or totalUsedTokens = totalAvailableForSale. So, line number 169 will always throw exception and finalize will never be possible.