CoinFabrik

Bricks4Us Token Audit

May 2018

By Coinfabrik

Introduction

Coinfabrik was asked to audit the contracts for the Bricks4Us token sale. Firstly, we will provide a summary of our discoveries and secondly, we will show the details of our findings.

Summary

The contracts audited are from the bricks4us-smart-contracts repository at https://github.com/Bricks4us/bricks4us-smart-contracts. The audit is based on the commit *535cf8c62a4cd11625e28a357ab5013f4b4c5c54*.

The audited contracts are:

- B4UCrowdsale-flat.sol: Crowdsale logic with buy functions.
- B4U-flat.sol: The token itself

The Bricks4Us token sale will have an initial supply of 100000000 tokens divisible up to 18 decimals. Some minor issues were found which could affect future maintainability, and a possible false positive. However, none of these are severe, but we still recommend implementing some changes to avoid future breakage.

The following analyses were performed:

- Misuse of the different call methods: call.value(), send() and transfer().
- Integer rounding errors, overflow, underflow and related usage of SafeMath functions.
- Old compiler version pragmas.
- Race conditions such as reentrancy attacks or front running.
- Misuse of block timestamps, assuming anything other than them being strictly increasing.
- Contract softlocking attacks (DoS).
- Potential gas cost of functions being over the gas limit.
- Missing function qualifiers and their misuse.
- Fallback functions with a higher gas cost than the one that a transfer or send call allows.
- Fraudulent or erroneous code.
- Code and contract interaction complexity.
- Wrong or missing error handling.
- Overuse of transfers in a single transaction instead of using withdrawal patterns.
- Insufficient analysis of the function input requirements.

Detailed findings

Minor severity

Possible false positive in checkPurchaseAllowed

There exists the possibility of checkPurchaseAllowed giving off a false positive at the B4UCrowdsale-flat.sol contract. If it gets called with a great enough number, the value resulting from executing the second term of the following predicate could overflow:

```
        return (now<phases[phaseIndex].endTime) && ((sold + purchaseTokenAmount) <= limit);
```

This would mean that getTokenAmount returns a value that is barely enough to overflow, which could get rejected when updating *investedTokenOf[msg.sender]*. A way to fix this would be by rewriting the return statement into something like the following:

```
        return (now<phases[phaseIndex].endTime) && (purchaseTokenAmount <=
purchasetokenAmount + sold) && ((sold + purchaseTokenAmount) <= limit);
```

This issue is not severe, though, as the mistaken value gets reverted when calling the safeMath function in the following line:

```
        investedTokenOf[msg.sender] = investedTokenOf[msg.sender].add(tokens);
```

However, this could cause the inutilization of the contract for a buyer meeting the condition until the phase gets changed.

Enhancements

Possible refactoring of hasEnded()

In B4UCrowdsale-flat.sol it'd be possible to refactor the *hasEnded()* function, which calls *getCurrentPhaseIndex()* with an ad hoc parameter of 1. It'd be possible to make it have a *require* clause for that case in particular. It could possibly be rewritten as:

```
                uint256 phaseIndex = getCurrentPhaseIndex();
                uint256 sold = phases[phaseIndex].phaseTokensSold;
                return (now < phases[phaseIndex].endTime && sold < limit);
```

This would help improve maintainability, since the resultant code is more legible and doesn't have hardcoded parameters

Usage of current compiler pragmas

The contracts' code use the following compiler pragma:

```
pragma solidity ^0.4.18;
```

It's advised for the development team to update such it and use newer compiler features, like having constructors declared with the *constructor* modifier and emitting events through the usage of the *emit* keyword, since they improve maintenance and legibility of the contract code.

Conclusion

We found the contracts to have some minor problems, which are easily fixable. Overall they were well documented and had no serious issues. The development team might want to reconsider on the remarks we've made, especially the possibility of a false positive, which isn't considered likely but could happen. We also found the contract to be well documented and easy to follow. Conditions for inputs were found to be adequate, and phase addition was found to require chronological order, which is necessary for maintaining coherence when calling *getCurrentPhaseIndex()*.