Inbest Token Audit

April 2018

By Coinfabrik

Introduction

Coinfabrik haswas asked to audit the contracts for the Inbest 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 inBest-token repository at https://github.com/inBest-today/inbest-token. The audit is based on the commit *2951db7017f9b53d83c8566454608a495e70f09d*.

The audited contracts and the sha256 sum of their contents are:

```
59be60631ce8eb5f1b22e926f597687b2bef90ac134162e1f8ceb5516bfe07e3 InbestDistribution.sol
7129e39cae2b1f6addfc99f4d4ff42e2d37006db9fffc343d7172e84496019a2 InbestToken.sol
```

The Inbest token sale will have a total of 1 million tokens issued. Half of it will be distributed during the presale period.

No actual security concerns were found. Nevertheless, we recommend updating compiler pragmas.

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 higher gas cost than a what 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

Enhancements

Use of outdated solidity compiler pragmas

As of 0.4.21, it is encouraged to use emit while using events, since it improves code legibility and helps differentiate between an event emission and a function call. This is not done in any of the event emission statements. We recommend developers to include newer features like defining constructors using `constructor(...){...}` instead of naming them, or explicitly emitting events.

Possible race condition on approval

It is known that an attack could be done on ERC20 contracts whereby a user could use the *transferFrom* method to send himself more tokens than what he was approved to send. This can happen because a transaction sending tokens to himself could be mined before a change in the amount of tokens was made effective. More details can be found in this paper. This can easily be solved by overriding the StandardToken method *approve* with the following:

```
function approve(address spender, uint256 value) public returns (bool) {
        require (value == 0 || allowed[msg.sender][spender] == 0);
        allowed[msg.sender][_spender] = value;
        //...
}
```

While the zeppelin contracts already have *increaseApproval* and *decreaseApproval* methods, it is best to avoid mistakes in the handling of these functions.

Observations

We write here some of the verifications done to the contracts to indicate that they were performed even when no issue was found during the audit.

- Misuse of the different call methods: *call.value()*, *send()* and *transfer()*.
  We found no incorrect use of *call()* or *send()*, and *transfer()*.
- Integer rounding errors, overflow, underflow and related usage of SafeMath functions.
  The mathematical operations performed in BaseToken.sol are protected against overflow using requires for input parameters.
- Race conditions such as reentrancy attacks or front running.
  There are three possible calls to another contract:
    - When checking that the total supply in the token contract is the same as the available total supply, calling a getter.

    - When transferring tokens, both when claiming the allocation for an address and when transferring from the company allocation. This is done using the token's transfer function so it is safe from reentrancy attacks.
- Misuse of block timestamps, assuming things other than them being strictly increasing.
  Timestamps are used to verify that the start time for the sale is in the future. While it is possible for allocations to be locked if a malicious miner sets the timestamp as greater than the starting time before it is necessary, such an attack is not probable due to the fact that a desynchronized node can't build consensus. This could be completely mitigated by setting the start time a few hours later than the deployment of the contract and setting the allocations right away.
- Contract softlocking attacks (DoS) / unbounded gas usage.
  No function in the contract has a loop that can be abused to cause a soft lock or an unbounded usage of gas.

Conclusion

We found the contracts to be simple and straightforward and to have an adequate amount of documentation. No urgent security concerns were found with the code, but we advice implementing the changes proposed in this document. We found the contracts to use well tested libraries and the devs assured that input precondition were made explicit.

References

- "ERC20 API: An Attack Vector on Approve/TransferFrom Methods" https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/