SMART CONTRACT AUDIT REPORT

For

Abele Token (Order #18AUG2019)

Prepared By: Chandan Kumar

Prepared For: Abele Group

Prepared on: 18/08/2019

audit@etherauthority.io

Table of Content

1. Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

2. Overview of the audit

The project has two main smart contract files:

- AbeleToken.sol
- AbeleTokenSale.sol

It contains approx **572** lines of Solidity code. All the functions and state variables are well commented, logical approach of coding is very neat and clean, and taken care of required security measures, but it contains cross version code approach, which was bound to fail with any compiler version, so it was not in state to attempt advance test for audit, so auditor made some minimal essential changes to apply audit test of next level , those reasons and changes are as below.

Reasons:

- Code was split into multiple files with structured folders.
- Solidity version is too old.
- Inconsistent version across files.
- Code block mismatch and conflict with version specified
- Inheritance conflict

Changes:

- All files merged into one for better readability and compact compiled output.
- Specified version kept to 0.4.19 (which was max across all files)
- All codes blocks mismatching with above version, changed to match with version
- Inheritance conflict removed
- Input parameter applied on constructor, for version discipline, converted to onlyOwner type public function, (because it was not called from any other part of contract.)

The audit was performed by two senior solidity auditors at EtherAuthority. The team has extensive work experience in developing and auditing the smart contracts.

This audit also checked the business data provided in the document by Abele Group.

This audit procedure also included the use of automated software to further scan of the code to identify potential issues:

For example:

https://tool.smartdec.net/scan/e704269ef51e4a3d92f7d9b04d4a9edc

https://mythx.io tool provided as remix.ethereum.org plugin

Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version is old	Not Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Moderated
	Function access control lacks management	Passed
	Critical operation lacks event log	Moderated
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Not Passed
Code Specification	Visibility not explicitly declared	Not Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Not Passed
	Other code specification issues	Passed
Gas Optimization	Assert() misuse	Not Passed

	High consumption ‘for/while’ loop	N/A
	High consumption ‘storage’ storage	Passed
	“Out of Gas” Attack	Passed
Business Risk	The maximum limit for mintage not set	N/A
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: **NOT PASSED**

3. Attacks tested on the contract

In order to check for the security of the contract, we tested several attacks on the code. Some of those are as below:

3.1: Over and under flows

SafeMath library is used in the contract, which prevented the possibility of overflow and underflow attacks.

3.2: Short address attack

Although this contract **is not vulnerable** to this attack, it is highly recommended to call functions after checking the validity of the address from the outside client.

3.3: Visibility & Delegatecall

Delegatecall is not used in the contract thus it does not have this vulnerability. And visibility is also used properly.

3.4: Reentrancy / TheDAO hack

Use of “require” function and Checks-Effects-Interactions pattern in this smart contract mitigated this vulnerability.

3.5: Forcing ether to a contract

Here, the Smart Contract’s balance has never been used as guard, which mitigated this vulnerability

3.6: Denial Of Service (DoS)

There **is No** any process consuming loops in the contracts which can be used for DoS attacks. and thus this contract is not prone to DoS.

4. Good things in the smart contract

4.1 Checks-Effects-Interactions pattern

While transferring tokens, this contract does all the process first and then transfers them. The same while doing other process too. This is very good practice which prevents malicious possibility. For example: transfer() function.

4.2 Functions input parameters passed

The functions in this contract verifies the validity of the input parameters, and this validations cannot be by-passed in anyway.

4.3 Conditions validations

```
function transfer(address _to, uint256 _value) public returns (bool) {
  require(_to != address(0));

  // SafeMath.sub will throw if there is not enough balance.
  balances[msg.sender] = balances[msg.sender].sub(_value);
  balances[_to] = balances[_to].add(_value);
  Transfer(msg.sender, _to, _value);
  return true;
}
```

The validation of input parameters are not done to prevent overflow and underflow of integers. Although use of SafeMath library also would be good programming flow.
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol

4.3 Variables defined as constant

```
// Constants
string  public constant name = "Abele Token";
string  public constant symbol = "ABELE";
uint8   public constant decimals = 5;
uint256 public constant INITIAL_SUPPLY      = 3000000000000;
uint256 public constant CROWDSALE_ALLOWANCE = 1000000000000;
uint256 public constant ADMIN_ALLOWANCE     = 2000000000000;
```

This is a good thing as it consumes less space in the memory.

5. Critical vulnerabilities found in the contract

Critical issues that could damage heavily the integrity of the contract. Some bug that would allow attackers to steal ether is a critical issue.

5.1 No automated token issuance - AbeleTokenSale.sol

```
function () public payable whenNotPaused beforeDeadline afterStartTime saleNotClosed
    require(msg.value >= minContribution);

    // Update the sender's balance of wei contributed and the amount raised
    uint amount = msg.value;
    uint currentBalance = balanceOf[msg.sender];
    balanceOf[msg.sender] = currentBalance.add(amount);
    amountRaised = amountRaised.add(amount);
}
```

balanceOf variable which is being updated is a local variable of the AbeleTokenSale contract. It does not issue real Abele Tokens from its own independent contract.

Also, ETH_RATE is defined but never used. Use it appropreately.

Resolution:

Keep both token and crowdsale contracts separate. In the crowdsale contract, just implement the interface of token contract and do all the token transfer and other needed processes in the token contract. So, it would be contract to contract communication.

Another option is to keep only one contract for token contrat and implement crowdsale features into itself.

5.2 tokenReward variable is not initiated - AbeleTokenSale.sol

```
// The token being sold
AbeleToken public tokenReward;
```

This variable never initiated. Ideally that should be done in constructor function or create another function which can be called by owner and you need to specify the token contract address to initiate this variable.

6. Medium vulnerabilities found in the contract

Those vulnerabilities that could damage the contract but with some kind of limitations. Like a bug allowing people to modify a random variable.

6.1: Use require condition in place of Assert - SafeMath Library

For any situation when this safemath will fail, then in that situation assert will consume all the maximum gas.

This will give “surprise charges” to users who had reverted transactions and had burned a lot of ether as a gas cost.

Require only consumes whatever gas used. So, require is always cheaper than assert. Avoid using assert unless really necessary!

6.2: Non-initialized return value - AbeleToken.sol contract

transfer() function doesn't initialize return value. As a result, default value will be returned.

Many DEX uses return value of transfer function to determine if token transfer were done or not.

If transfer function does not return correct value, then your token would not be compliant to many DEX you may wish to list in the futhre!

7. Low severity vulnerabilities found

Those do not damage the contract, but better to resolve and make code clean.

7.1: Compiler version can be fixed - both contracts

The contract has lower solidity version than the current one. This version gap is quite high in contract and there were many improvements afterwards.

So, it is good practice to deploy the contract having latest solidity version. The solidity version at a time of audit is: 0.5.11

7.2: Deprecated elements – AbeleTokenSale.sol contract

The way constructor function was defined is conflict with the version. You need to use “constructor” keyword to define constructor function.

And many other deprecated code attempts exist in contract.

7.3: Missing approval of ownership transfer:

This may be difficult to get control back if by mistake transferred to the wrong address.

8. Very low severity vulnerabilities found

The presence of these things does not make any negative effect. But just to clean up the code.

8.1: No explicit visibility - AbleTokenSale contract

Visibility is not specified at line #161, #122. Please note that this is not a big issue as it takes default to *“public”*. But it's suggested to explicitly define visibility to avoid confusion.

8.2: Unused Interface elements - AbleTokenSale contract

The contract ERC20, at line #16 has unused elements. No other elements are used except transfer() function. so it’s better to remove them as not used anywhere.

9. Gas Optimization Discussion

=> The Contract is most optimum for the gas cost. There is no gas expensive loops, or logical unnecessary processes.

10. Discussions and improvements

10.1 approve() of ERC20 Standard

To prevent attack vectors regarding approve() like the one described here: https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit , clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. THOUGH the contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before

10.2 Consider adding ownership contracts

The Openzepelin ownable contract has default issue. The ownable contracts much implement the “accepting ownership” logic when transferring the ownership. This prevents sending ownership to incorrect contracts as we seen that ruined many contracts!

10.3 Consider adding Safeguard function

In any unexpected events, owner of the contract can put safeguard (halt token movement). Once the problem is resolved, then the owner can lift the safeguard and everything comes back to normal.

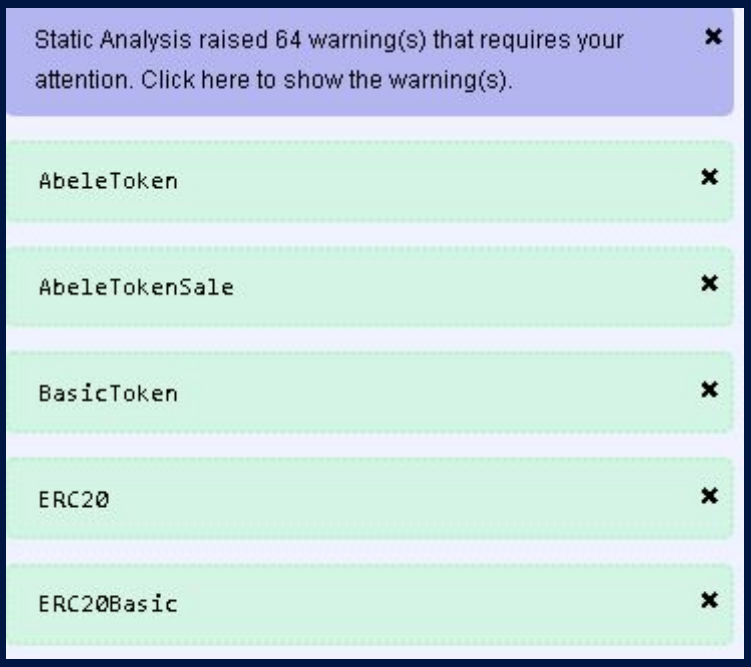
10.4 No instant token distribution

This vesting (crowdsale) contract does not make instant distribution of tokens when someone sends ether to the contract address. as well as no proper events in fallback function just to log all the ICO deposits.

11. Summary of the Audit

Overall, the code is ERC20 token implementation as well as vesting for ICO/IEO.

Compiler showed couple of warnings, as below:



Now, we checked that the warnings in purple division, are due to their static analysis, which includes like gas estimations and all. So, it is important to supply correct gas values while calling various functions.

Those warnings can be safely ignored as should be taken care while calling the smart contract functions.

Please try to check the address and value of token externally before sending to the solidity code.

It is also encouraged to run bug bounty program and let community help to further polish the code to perfection.