28 MAY 2018

AIRBLOC
SMART CONTRACT AUDIT REPORT 1

HAECHILABS

01. INTRODUCTION

This report audits the security of ABL ERC20 token and the first pre-sale smart contract created by the AIRBLOC team. HAECHI Labs audited the smart contract code produced by the AIRBLOC team to ensure that it was designed for the purposes outlined in the white paper and other published materials, and that the code was secure.

The code used for this audit can be found in the "AIRBLOC / token" Github repository (https://github.com/AIRBLOC/token). The final commit of the code used for the audit is "dd150e02317b29cfa07b04cf8f36766a34972076". Out of the code, "ABLG.sol", "ABLGExchanger.sol", "OwnedTokenTimelock.sol", and "openzepplin-solidity", which are open source libraries, are not subject to this audit.

The AIRBLOC team's white paper can be found at the link.

02. AUDITED FILE

- ABL.sol
- PresaleFirst.sol
- OwnableToken.sol

03. ABOUT "HAECHI LABS"

"HAECHI Labs" has a vision to contribute to a healthy blockchain ecosystem through technology. HAECHI Labs conducts in-depth research on the security of smart contracts. There has been many incidents caused by smart contract vulnerabilities including The DAO, Parity multisig wallet and SmartMesh (ERC20) hacking. "HAECHI Labs" is committed to designing and implementing the security auditing of smart contracts. We provide smart contract related services to enable customers to optimize the gas cost during operation and to implement a secure smart contract that meets their purpose. "HAECHI Labs" is composed of people who have experiences as software engineers in start-ups and lead blockchain researchers at Decipher (the blockchain research group at Seoul National University) and nonce research.

04. ISSUES FOUND

The issues found are divided into 'major' and 'minor' depending on their importance. Major issues should be modified since the code is neither secure nor implemented for its original intention. Minor issues would be potentially problematic and require modification. HAECHI Labs recommends the AIRBLOC team to improve on every issue found.

Major Issues

1. The amount of ABL token issued is different from what is specified in the white paper.

```
// Token Distribution Rate
uint256 public constant SUM = 400000000;   // totalSupply
uint256 public constant DISTRIBUTION = 221450000; // distribution
uint256 public constant DEVELOPERS = 178550000;   // developer

// Token Information
string public name = "Airbloc Token";
string public symbol = "ABL";
uint256 public decimals = 18;
uint256 public totalSupply = SUM;
```

ABL.sol

The total amount of ABL token issued is 400000000 (*SUM*), the amount of token distributed through the token sale is 221450000 (*DISTRIBUTION*), and the amount of distributed to the development team is 178550000. However, because the decimals value is 18, *totalSupply* must be *400000000 * (10 ** uint256 (decimals))* instead of 400000000 corresponding to . To correct this, it is better to multiply *SUM*, *DISTRIBUTION*, and *DEVELOPERS* by *10 ** uint256 (decimals)*, respectively.

2. The exchange formula of ETH received from PresaleFirst.sol to ABL token is incorrect.

```
// buy
uint256 tokenAmount = purchase.div(1000000000000000000).mul(rate);
maxcap = maxcap.sub(purchase);

```

PresaleFirst.sol

Investors will deposit ETH into the pre-sale contract and receive ABL tokens at the pre-determined exchange rate. However, the pre-sale primary exchange rate stated on the white paper is different from what is described in the code. On the white paper, investors participating in the first pre-sale can receive 11500ABL per 1 ETH. In PresaleFirst.sol, however, investors can receive *purchase.div(1000000000000000).mul(rate)*. Since 1ETH = 10 ^ 18 wei and decimals are 18, the correct ABL token exchange formula is purchase.mul(rate).

Minor Issues

1. The intention of implementing *modifier locked* could lead to a misunderstanding.

```
    bool isLocked = true;

    modifier locked() {
        require(!isLocked);
        _;
    }

    function unlock() public onlyOwner {
        isLocked = false;
    }

    function lock() public onlyOwner {
        isLocked = true;
    }

    function transfer(address _to, uint256 _value) public locked returns (bool) {
        return super.transfer(_to, _value);
    }
```
ABL.sol

There is a *locked modifier* in the *transfer* function in ABL token, which can prevent token transmission according to the *isLocked* value. This can be deemed suspicious. Even when ABL token sales period is over, the smart contract owner can lock and exploit the token transfer at any time. For example, when listing the token on an exchange, the development team can move the tokens in advance and temporarily lock the token transfer to prevent investors from pumping and dumping its tokens. Therefore, we recommend to delete this lock function in transfer.

If the team is concerned about tokens being exchanged and sold during the token sale period, they could implement one of the following two options: The first is to distribute tokens after the token sale period. The other is to force the development team not to lock the code once the code is unlocked.

2. The fallback function of PresaleFirst.sol would be suitable for *external* rather than *public*.

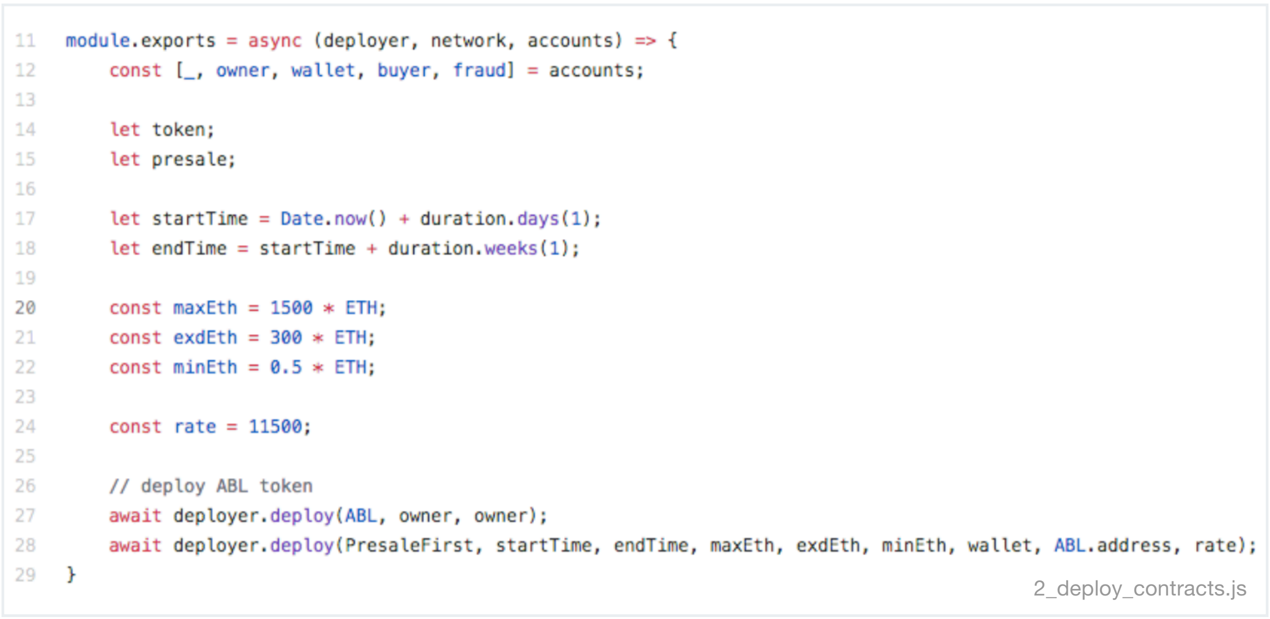
```
    function () public payable {
        collect(msg.sender);
    }
```
PresaleFirst.sol

The *function () public payable*, fallback function, in PreSaleFirst would be suitable for *external* rather than *public*. *public* can call both *internal* and *external* and it consumes more gas than *external*. It is better to use *external* rather than *public*, because the function is only called externally by investors during the token purchase.

3. Reducing *maxcap* in PresaleFirst.sol makes it difficult to trace the initial cap.

The *maxcap* is the amount of ETH available during the pre-sale. PresaleFirst reduces the total *maxcap* by subtracting the deposited amount of ETH from *maxcap* each time the token sale proceeds. However, in this case, *maxcap* is constantly changing, making it difficult to trace whether the correct *maxcap* is initially set. Therefore, it is a good idea to set *maxcap* as a constant variable and a new variable like *weiRaised* to check whether *weiRaised* exceeds maxcap.

4. The pre-sale period is not explicit.



```
11  module.exports = async (deployer, network, accounts) => {
12      const [_, owner, wallet, buyer, fraud] = accounts;
13
14      let token;
15      let presale;
16
17      let startTime = Date.now() + duration.days(1);
18      let endTime = startTime + duration.weeks(1);
19
20      const maxEth = 1500 * ETH;
21      const exdEth = 300 * ETH;
22      const minEth = 0.5 * ETH;
23
24      const rate = 11500;
25
26      // deploy ABL token
27      await deployer.deploy(ABL, owner, owner);
28      await deployer.deploy(PresaleFirst, startTime, endTime, maxEth, exdEth, minEth, wallet, ABL.address, rate);
29  }
```

2_deploy_contracts.js

The *startTime* and *endTime* of PresaleFirst are determined by a constructor above. *startTime* and *endTime*, injected from the "2_deploy_contracts.js" smart contract deployment script, may look different from the token sale period. The lines 17-18 of the image above set the start time of the sale through *Date.now()*. In this case, the distribution script should be executed the exactly 24 hours before the pre-sale to match the period specified on the website. We recommend to set the unix epoch timestamp as a constant to designate when the token sale starts and ends.

5. If you can withdraw ETH during the pre-sale, it might seem suspicious.

```
64      function collect(address _buyer) public payable onlyWhitelisted whenNotPaused {
65          require(_buyer != address(0));
66
67          require(preValidation());
68
69          uint256 refund;
70          uint256 purchase;
71
72          (refund, purchase) = getTokenAmount(_buyer);
73          emit TokenAmount(_buyer, refund, purchase);
74
75          // refund
76          _buyer.transfer(refund);
77
78          // buy
79          uint256 tokenAmount = purchase.div(1000000000000000).mul(rate);
80          maxcap = maxcap.sub(purchase);
81
82          // wallet
83          wallet.transfer(purchase);
84          buyers[_buyer].FundAmount = buyers[_buyer].FundAmount.add(purchase);
85          buyers[_buyer].TokenAmount = buyers[_buyer].TokenAmount.add(tokenAmount);
86          emit BuyToken(_buyer, purchase, tokenAmount);
87      }
```
PresaleFirst.sol

If ETH is withdrawn during the pre-sale, it can be re-deposited to the pre-sale smart contract. This procedure can be repeated and be exploited to increase the amount of tokens collected and to fill *maxcap* without any actual investment.

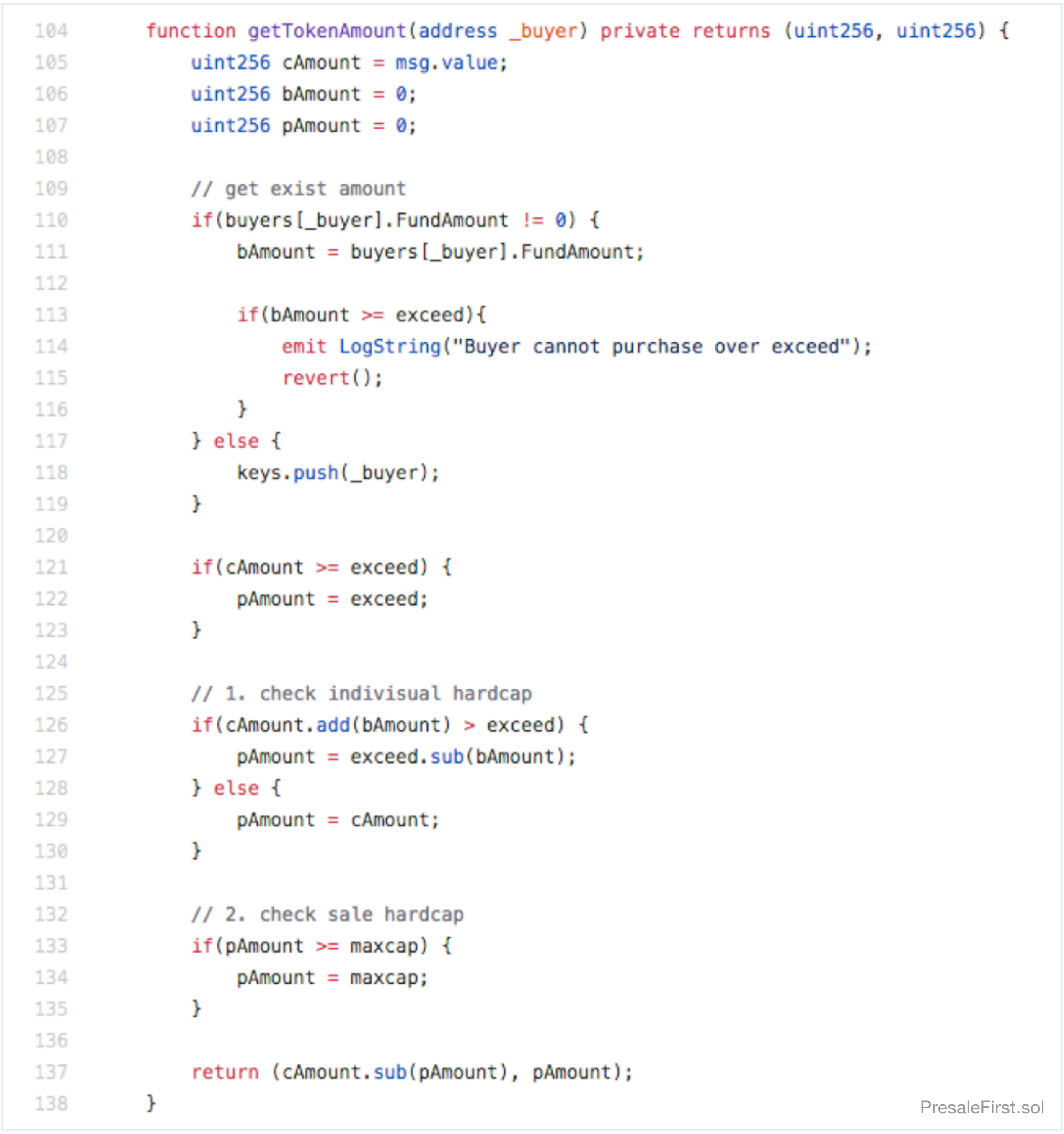
The current code is the 83rd line in the image above. When an investor deposits 10 ETH, it is sent to the other wallet and the deposit is withdrawn. For example, if you send withdrawals, 10 ETH, to the presales smart contract again, the smart contract logs 20 ETH deposits. This will allow you to fill up the *maxcap* with a small amount of ETH and receive a large amount of ABL tokens with a small amount of ETH. Later, when AIRBLOC will release ABL tokens after the end of pre-sale, we recommend AIRBLOC to code to move ETH from the smart contract to investors' wallet.

6. Gas is wasted in PresaleFirst.sol's *Buyer* struct storing unnecessary data.

```
    struct Buyer {
        uint256 FundAmount;
        uint256 TokenAmount;
    }
```
PresaleFirst.sol

There are *FundAmount* and *TokenAmount* in the *Buyer* struct, and *TokenAmount* is a value that can be obtained using *FundAmount * rate*. Therefore, *TokenAmount* can be obtained even if *TokenAmount* is not saved. This unnecessary data storage leads to waste of gas.

7. The function of *getTokenAmount* function of PresaleFirst.sol needs to be separated.



```
    function getTokenAmount(address _buyer) private returns (uint256, uint256) {
        uint256 cAmount = msg.value;
        uint256 bAmount = 0;
        uint256 pAmount = 0;

        // get exist amount
        if(buyers[_buyer].FundAmount != 0) {
            bAmount = buyers[_buyer].FundAmount;

            if(bAmount >= exceed){
                emit LogString("Buyer cannot purchase over exceed");
                revert();
            }
        } else {
            keys.push(_buyer);
        }

        if(cAmount >= exceed) {
            pAmount = exceed;
        }

        // 1. check indivisual hardcap
        if(cAmount.add(bAmount) > exceed) {
            pAmount = exceed.sub(bAmount);
        } else {
            pAmount = cAmount;
        }

        // 2. check sale hardcap
        if(pAmount >= maxcap) {
            pAmount = maxcap;
        }

        return (cAmount.sub(pAmount), pAmount);
    }
```

PresaleFirst.sol

The *getTokenAmount* function returns two types of variables. First, it returns the amount of ETH which is used to purchase ABL tokens. Second, it returns the amount of ETH, which is not used to purchase ABL tokens, but which is the amount of refund by checking the individual cap and pre-sale cap.

However, the operation logic of the function is complicated. As a rule of thumb, the functions of smart contracts should be simple. If you write complex code, you are more likely to get errors. The function of *getTokenAmount* has too many logics described above, therefore, we recommend to modularize and keep the function simple by separating them from each other.

In addition, the logic of function is complicated and there are some inefficiencies such as calling the *revert* function during the operation. We recommend to use *require* on top of *getTokenAmount* function to inform the failure as soon as possible rather than *revert* during operation by comparing *buyers[_buyer].FundAmount* and *exceed* value on lines 111 to 115.

8. *LogString* Event does not always occur.

In the *getTokenAmount* function described in "Minor Issue 7", line 114 generates a *LogString* event. However, *LogString* does not always occur because of the *revert* function on line 115. Therefore, we recommend to delete the unnecessary code.

9. It is better to use *Whitelist* instead of *OwnableToken*.

The *OwnableToken* are designed in such a way that the owner can specify a new owner. This may be dangerous because there can be too many owners. And once you become an owner, you will not lose your rights until you transfer the owner through *transferOwnership*. Therefore, if there is a malicious owner, there is no way to stop malicious owner's attack. If the AIRBLOC team wants only those who are chosen to call a particular feature, we recommend to use *Whitelist* in the "openzeppelin-solidity" library.

10. Check the transfer lock state before doing the *release* function of PresaleFirst.sol

```
155     function release() public onlyOwner onlyOnce {
156         require(block.timestamp > endTime);
157         once = true;
158
159         for(uint256 i = 0; i < keys.length; i++) {
160             token.safeTransfer(keys[i], buyers[keys[i]].TokenAmount);
161             emit Release(keys[i], buyers[keys[i]].TokenAmount);
162         }
163     }
```

PresaleFirst.sol

It is better to remove the lock on the *transfer* function of ABL token. However, if there is a lock on the transfer, it is recommended that you use *require*, on the top of the *release* function to distribute ABL token after the pre-sale, to see if the transfer is locked. The current *release* function will fail if ABL tokens are locked when executing ABL token *transfer* function on line 160. It is important to check the conditions before executing the logic of the function because it is expensive to run a smart contract logic on Ethereum.

11. The *release* function of PresaleFirst.sol is suitable for *external* rather than *public*.

If the *release* function is declared as *public* in the minor issue 10, it can be called both *internal* and *external*. However, since the function will only be called externally by the development team after the pre-sale, we recommend to use *external* rather than *public*. Also, *external* consumes less gas than *public*.

12. It is better to subtract token after transferring the token from the *release* function of PresaleFirst.sol.

If the external call calls the *release* function again, the re-entrancy problem may occur. The re-entrancy issue is to transfer the token to the same address repeatedly due to external call calling the *release* function. This happens because the amount of tokens sent to buyer is not set to zero. To solve this problem, we recommend to set *tokenAmount* to zero before sending tokens. Since the token is ABL tokens created by the AIRBLOC team, so it can be considered as a trustworthy smart contract. In addition, there is the *onlyOnce* implementation to prevent reentrancy, so there may not be an immediate problem. However, it is a good idea to focus on safe coding patterns in general. Therefore, we recommend to change the token amount to zero from *mapping buyer* before sending ABL tokens on the *release* function in the minor issue 10

05. DISCLAIMER

The findings of this report is not exhaustive and is limited to the current understanding of known security patterns. This report only discusses known technical issues and does not include any investment advice, the profitability of the business model nor any other issues outside the technical realm. Regardless of the problems described in this report, there may be unknown problems and defects in Ethereum or the solidity language. In order to create a secure smart contract, we recommend the AIRBLOC team to fix the problems proposed in this report and conduct enough testing.