SOLIDIFIED

Audit Report for DADI. January 28, 2018.

Summary

Audit Report prepared by Solidified for DADI covering the Dadi Public Sale contract.

Process and Delivery

Two (2) independent Solidified experts performed an unbiased and isolated audit of the below token sale. The debrief took place on January 28, 2018 and the final results are presented here.

Audited Files

The following files were covered during the audit:

- DadiPublicSale.sol

Intended Behavior

The provided intended behavior spec can be found here: https://docs.google.com/document/d/1iQZIn6X6G5C2pV_Ua0MO0T_bLN_myBWX__R8sAcUbEQ/edit?usp=sharing

Issues Found

1. Sale state can be changed arbitrarily

The documented sale states implies a rigid linearity between states (Preparing happens before PublicSale, that happens before PublicSaleFinalized, and so forth), but the function setState() allows to change it arbitrarily. This mechanic allows the owner to manipulate the sale in various ways, such as bypass the individual cap for a specific address.

How it works:

Suppose address "0x111" makes an purchase that matches is individual cap, that grants him T amount of tokens. After the transaction is mined, the owner call the function setState() to TokenDistribution. After that, he calls distributeTokens to address "0x111" and then makes a

last transaction to revert the state back to PublicSale. Now, "0x111" can make another transaction and bypass it's personal cap.

Recommendation:

Change the setState function to a updateState function, where it only allows the sale to go to the next state and forbids to return to a previous one. Example implementation:

```
    function updateState () public onlyOwner {
        state = SaleState(uint256(uint256(state) + 1));
        LogStateChange(state);
    }
```

2. Non-enforceable state

Related to issue 1, most of the contract functions are not enforcing the correct state, and therefore can be called in any state, also allowing the state to be manipulated back and forth.

Recommendation:

Add requirements in state dependent functions to only succeed if sale is in the correct state. Consider adding these functions:

startPublicSale()	requires	state = Preparing
offlineTransaction()	requires	state = PublicSale
finalizeSale()	requires	state = FinalizedSale
setTokenDistribution()	requires	state = Success
redistributeTokens()	requires	state = TokenDistribution

3. Tokens are calculated in the wrong unit

The function calculateTokens() considers the parameter _amount as being in ETH, but it's actually being called with WEI, increasing the amount of tokens by 18 decimal cases.

Recommendation:

To make this calculation correct, is necessary to get the variable ethRate in the correct unit(weiRate). See suggestion 10.

4. Sales Wallet is not implemented properly

DadiPublicSale.sol / Line 232

There is no option to remove the added address from `saleWallets` array. If any address is added accidently it cannot be removed and can potentially lead to the loss of funds. It has a small chance of occurrence, but is a potential threat to the funds.
Also, there is a chance of duplicate wallet entry. This has to be taken care of.

Recommendation:

This threat can be avoided by adding an option to remove a particular address from the `saleWallets` array. Removing an element from the array can be expensive, but it is the tradeoff that you can make to potential loss of funds. Use of mapping can also be considered.
Also consider adding a getter method private to the owner to verify the list. Currently, since it is a private array and no getters are associated with it, there is no way to check for the list of addresses that are present.

```
function getWalletAddress() external onlyOwner view returns (address[]) {
      return saleWallets;
}
```

5. Empty balances mapping can revert the transaction during token distribution

Balances mapping in the StandardToken is empty during the token distribution. It can cause the distribution to fail during token transfer.

Recommendation:

This can be resolved by adding the total token supply to the mapping:

```
token.transfer(_address, tokens);
```

6. Chance of funds getting locked up after the sale is closed

If there is no wallet address present in the `saleWallets` array during the crowdsale, the funds collected will still be locked in the contract. Owner will not know if the saleWallet array is empty during the token buying process. There is no option to retrieve the funds once the sale is closed.

Recommendation:

This can be solved with any one of the following methods:

1. Modify the existing `forwardFunds` to allow access for the owner so that it can be called by the owner manually.

```
function forwardFunds (uint256 _value) public onlyOwner {
        uint accountNumber;
        address account;

        // move funds to a random saleWallet
        if (saleWallets.length > 0) {
           accountNumber = getRandom(saleWallets.length) - 1;
           account = saleWallets[accountNumber];
           account.transfer(_value);
           LogFundTransfer(account, _value);
        }
}
```

2. Add an address to the saleWallets array during the contract initialization.

```
function DadiPublicSale (StandardToken _token, uint256 _tokenSupply) public {
        require(_token != address(0));
        require(_tokenSupply != 0);
        token = StandardToken(_token);
        tokenSupply = _tokenSupply * (uint256(10) ** 18);
        maxGasPrice = 60000000000; // 60 Gwei
        saleWallets.push(msg.sender); // Adding owner as one of the wallet address
}
```

7. Individual cap can be bypassed

If a buyer makes a combination of online and offline purchases he/she can bypass the defined individual cap. This happens because offlineTransaction() function does not check whether or not the buyer has already made any purchases.

Recommendation:

Add a check inside offlineTransaction() to guarantee that the buyer is not buying more than the individual cap allows.

8. Function redistributeTokens does not account for recipients tokens

This function remove tokens from the original investor and does not add them to the receiving investor mapping. This breaks an implicit invariant that the tokensPurchased must be equal to the sum of the investors tokens and opens the possibility for unaccounted tokens.

Recommendation:

Add the line in the body of redistributeTokens:
`investors[recipient].tokens = tokens;`

9. Function ethToUSD() misbehaves for amount lesser than 1 ETH

If the result of the multiplication (_amount * ethRate) is smaller than 10^18, this function will always return 0, because of the lack of floating points. In this case, it does not cause any harm, since if the amount is lesser than 1ETH for sure it's below the cap, but it's a non intended behavior nonetheless.

Recommendation:

To avoid this issue is necessary to have the variable ethRate in the correct granularity, representing how many usd 1 WEI equals to. See suggestion 10.

10. Consider changing ethRate unit

The ethRate reflects the dollar price per ether, but most of the contracts operations are made in WEI. Since the EVM can't handle decimal point, this choice impacts the calculations of rates lesser than 1 ETH, returning untruthful values. A better approach is to keep this variable in the smaller unit(eg. weiRate), making possible to execute more precision calculations, since there's no rounding in multiplications.

11. Older compiler version can lead to some bugs

Older compilers might be susceptible to bugs. It is always suggested to use the latest stable version. As of this writing the recommended version is 0.4.19. List of known compiler bugs and their severity can be found here (https://etherscan.io/solcbuginfo)

12. Use of older OpenZeppelin libraries

DadiPublicSale.sol / Lines 7 - 31

The ethRate reflects the dollar price per ether, but most of the contracts operations are made in WEI. Since the EVM can't handle decimal point, this choice impacts the calculations of rates lesser than 1 ETH, returning untruthful values. A better approach is to keep this variable in the smaller unit(eg. weiRate), making possible to execute more precision calculations, since there's no rounding in multiplications.

13. Install OpenZeppelin via NPM

SafeMath, Ownable, ERC20Basic, ERC20, BasicToken, and StandardToken were copied from the OpenZeppelin repository. OpenZeppelin’s MIT license requires the license and copyright notice to be included if its code is used, and makes it difficult and error-prone to update to a more recent version. Consider following the recommended way to use OpenZeppelin contracts, which is via the zeppelin-solidity NPM package. This allows for any bug fixes to be easily integrated into the codebase.

14. RedistributeTokens function is potentially dangerous

This function allows for the owner to redistribute any purchased token to another address, regardless of the investor will. This might reduce investors' trust in the contract, since they are susceptible to owner's action.

Recommendation:

Consider implementing an authorization mechanism to avoid arbitrary transfers.

15. Consider changing to a push mechanic

When the sale ends, investor only get bought tokens when the sale owner call distributeTokens function. Consider replacing this flow with a push mechanic, where investors make a call to withdrawal bought tokens. This has the benefit of saving a lot in gas cost and also shields buyers if owner goes missing(and never call this function).

16. Consider changing the way distributed tokens are accounted

To keep the accountability tight, consider not changing the following variables when distributing tokens:

```
investors[_address].tokens = 0;
investors[_address].contribution = 0;
```

This is bad, because it erases the link between buyers and the respective amount of tokens. A better approach is to add a new boolean variable in the Investor struct indicating whether or not tokens have been distributed.

17. Decimals into constant

Consider saving the decimal cases(10**18) into a constant. Right now it's a magic number that appears multiple times throughout the contract.

18. UpdateEthRate erroneous comment

The documentation regarding the return value of the updateEthRate function states that "Return true if the contract is in PartnerSale Period" but that doesn't not reflect the actual behavior of the contract.

19. Redundant owner declaration and assignment

DadiPublicSale.sol / Line 231, 278

```
owner = msg.sender;
```

Owner address declaration and assignment is already taken care by the Ownable contract. This redundant declaration and assignment can be removed from DadiPublicSale.

20. Redundant function declaration to get tokensPurchased

DadiPublicSale.sol / Line 492 - 494

```
function getTokensPurchased () public constant returns (uint256) {
        return tokensPurchased;
}
```

Public variables generate a getter by default and the function to retrieve the same can be removed.

21. Redundant msg.value check

DadiPublicSale.sol / Line 574

```
require(msg.value > 0);
```

nonZero modifier performs the msg.value check in the fallback function. The same check can be avoided in the buyTokens function

22. Use view/pure instead of constant

Make use of view/pure. Use view if your function does not modify storage and pure if it does not even read any state information

```
// Line 484
function getTokensAvailable () public constant returns (uint256)

// Line 501
function ethToUsd (uint256 _amount) public constant returns (uint256)

// Line 509
function getInvestorCount () public constant returns (uint count)

// Line 520
function getInvestor (address _address) public constant returns (uint256 contribu
tion, uint256 tokens, uint index)

// Line 530
function isInvested (address _address) internal constant returns (bool isIndeed)

// Line 605
function isValidContribution (address _address, uint256 _amount) internal constant
returns (bool valid)

// Line 614
function isBelowCap (uint256 _amount) internal constant returns (bool)

// Line 623
function getRandom(uint max) internal constant returns (uint randomNumber)
```

23. Use modifiers

Make use of modifiers for checking the sale status, msg.value and gas cost. Consider following the below structure for sale status

```
modifer saleShouldbe(SaleState _state) {
        require(state == _state);
        _;
}
function distributeTokens (address _address) public onlyOwner saleShouldbe(SaleSt
ate.TokenDistribution) returns (bool)
```

24. Make use of the SafeMath library

In places of arithmatic operations we can make use of the SafeMath library. It is not needed everywhere, but it is a good practive to follow the same pattern throughout the contract.

```
tokens = _amount * ethRate / tokenPrice;
tokens = _amount.mul(ethRate).div(tokenPrice);
```

25. Remove duplicate typecasting

Reduntant typecasting is performed in multiple places. This can be removed.

```
function setState (uint256 _state) public onlyOwner {
        state = SaleState(uint(_state));
        LogStateChange(state);
}
```

Closing Summary

Lots of issues were found during the audit, some of which are major and can break desired behaviour. It's strongly advised that these issues are corrected before moving on with the public sale. At the minimum, we recommend fixing all Major issues (1-6) and Minor (7-12).

Disclaimer

Audit Report for DADI. January 28, 2018.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Solidified platform from legal and financial liability.

Solidified Technologies Inc.