Enjin Token Sale Audit

AUTHOR: MATTHEW DI FERRANTE

2017-09-29

Audited Material Summary

The audit consists of the ENJToken and ENJCrowdFund contracts and their dependencies.

The contracts use standard ERC20 code and Ownable interfaces, along with SafeMath, and the two main contracts ENJToken and ENJCrowdFund are deployed separately, with the ENJCrowdFund given special permissions to the ENJToken contract.

ENJCrowdFund.sol

The ENJCrowdFund contract is responsible for the crowdsale process and is the contract that receives ether and updates respective balances inside the ENJToken contract.

It inherits from TokenHolder directly:

```
contract ENJCrowdfund is TokenHolder
```

TokenHolder inherits from Owned and Utils, which implement the Ownable pattern and some validation and SafeMath functions respectively.

The contract is well constructed and has no security issues, the biggest risk is argument mismatch when deliverPresaleTokens is called.

Constructor

```
function ENJCrowdfund(uint256 _totalPresaleTokensYetToAllocate, address
    _beneficiary)
validAddress(_beneficiary)
{
    totalPresaleTokensYetToAllocate = _totalPresaleTokensYetToAllocate;
    beneficiary = _beneficiary;
}
```

The constructor simply takes two arguments, tokens to be allocated and beneficiary, and registers them in the corresponding state variables. The beneficiary address is validated through validAddress - which ensures it is not the zero address.

setToken

```
function setToken(address _tokenAddress) validAddress(_tokenAddress)
    ownerOnly {
    require(tokenAddress == 0x0);
    tokenAddress = _tokenAddress;
    token = ENJToken(_tokenAddress);
}
```

This function can only be called once (will only run if Token is 0x00) and sets the ENJToken address that ENJCrowdfund will manipulate. Can only be called by the owner of ENJCrowdFund.

changeBeneficiary

```
function changeBeneficiary(address _newBeneficiary) validAddress(
    _newBeneficiary) ownerOnly {
    beneficiary = _newBeneficiary;
}
```

This function changes the beneficiary, or the receiver address for the ether raised by the crowdsale. Can only be called by the owner of ENJCrowdFund, and address must not be the 0x00 address.

```
function deliverPresaleTokens(address[] _batchOfAddresses, uint256[]
    _amountofENJ) external tokenIsSet ownerOnly returns (bool success) {
    require(now < startTime);
    for (uint256 i = 0; i < _batchOfAddresses.length; i++) {
        deliverPresaleTokenToClient(_batchOfAddresses[i], _amountofENJ[i])
            ;
    }
    return true;
}
```

This function simply calls deliverPresaleTokenToClient in a loop for each address:amount pair. It can only be called by the owner, and only if the Token address has already been set.

deliverPresaleTokenToClient

```
function deliverPresaleTokenToClient(address _accountHolder, uint256
    _amountofENJ) internal ownerOnly {
    require(totalPresaleTokensYetToAllocate > 0);
    token.transfer(_accountHolder, _amountofENJ);
    token.addToAllocation(_amountofENJ);
    totalPresaleTokensYetToAllocate = safeSub(
        totalPresaleTokensYetToAllocate, _amountofENJ);
    PresaleContribution(_accountHolder, _amountofENJ);
}
```

This function allocates presale tokens to a given client. It ensures that there are still tokens left to allocate, and transfers _amountofENJ tokens from the ENJCrowdFund address to the _accountHolder. It adds that amount through addToAllocation, substracts the added amount from totalPresaleTokensYetToAllocate, and fires off a PresaleContribution event.

It is internal and can only be called by the owner, and only through deliverPresaleTokens.

The subtraction is done through SafeMath which prevents underflows.

contributeETH

```
function contributeETH(address _to) public validAddress(_to) between
    tokenIsSet payable returns (uint256 amount) {
    return processContribution(_to);
}
```

This is just a forwarder to processContribution - ensures it can only be called while the sale is active, only once the Token is set, and validates the _to address.

processContribution

```
function processContribution(address _to) private returns (uint256 amount)
    {

    uint256 tokenAmount = getTotalAmountOfTokens(msg.value);
    beneficiary.transfer(msg.value);
    token.transfer(_to, tokenAmount);
    token.addToAllocation(tokenAmount);
    CrowdsaleContribution(_to, msg.value, tokenAmount);
    return tokenAmount;
}
```

This is a private function called only by contributeETH - it calculates tokens to be awarded from msg.value, transfers the Ether to the beneficiary, transfers the allocated tokens to the address _to, accounts for them with addToAllocation, fires a CrowdsaleContribution event, and returns the amount of tokens allocated.

```
function totalEnjSold() public constant returns(uint256 total) {
    return token.totalAllocated();
}
```

This function just returns the total tokens that have been allocated so far.

```
function getTotalAmountOfTokens(uint256 _contribution) public constant
    returns (uint256 amountOfTokens) {
    uint256 currentTokenRate = 0;
    if (now < week2Start) {
        return currentTokenRate = safeMul(_contribution, 6000);
    } else if (now < week3Start) {
        return currentTokenRate = safeMul(_contribution, 5000);
    } else if (now < week4Start) {
        return currentTokenRate = safeMul(_contribution, 4000);
    } else {
        return currentTokenRate = safeMul(_contribution, 3000);
    }
}
```

This function returns the amount of tokens that should be awarded for an Ether contribution. For the first week, it's wei_amount * 6000, from the 2nd week until the 3rd week, it's wei_amount * 5000, between the 3rd until the 4th week it's wei_amount * 4000, and after that it's wei_amount * 3000.

```
function() payable {
    contributeETH(msg.sender);
}
```

The fallback function is merely a forwarder to contributeETH, feeding msg.sender as the argument.

ENJToken.sol

The ENJToken contract implements the main logic for the ERC20 Enjin Token. It is a standard ERC20 Token, with a withdrawTokens function for tokens mistakenly sent to this address. It has no fallback function.

The contract's total supply is 1b, with 600m allocated to presale, 200m on crowdsale, 100m on incentivization, 50m on advisors, and 50m on the Enjin team. The decimals for the token is a standard 18 positions.

The crowdsale's end time is "Tue Oct 31 23:59:00 GMT 2017".

The contract inherits from ERC20Token and TokenHolder.

```
contract ENJToken is ERC20Token, TokenHolder
```

SafeMath is used on all token functions, through the Utils contract.

Security notes:

- The contract disables transfers until the sale is over, and until the tokens are "released" to the public - but this can be worked around with a wrapper contract, if a user buy a large percentage of the crowdsale and assigns it to this ENJWrapper contract, the contract holds the tokens and can resell them by issuing "token promises" that are redeemable once the real ENJToken's tranfers are unlocked. In general, locking up tokens for a long time is not feasible on Ethereum.
- The TokenHolder contract contains a withdrawTokens function:

```
function withdrawTokens(IERC20Token _token, address _to, uint256
    _amount)
public
ownerOnly
validAddress(_token)
validAddress(_to)
notThis(_to)
{
assert(_token.transfer(_to, _amount));
}
```

This function allows the contract owner to collect tokens mistakenly sent to this contract, however it does not deal with the mistakenly sent ether use case. While the contract has no default payable, and hence will reject eether sent through calls, it can still accept ether *assigned* to the contract through a selfdestruct(ENJToken) execution. This is of course not likely to be done by mistake, but still a possibility.

Constructor

```
function ENJToken(address _crowdFundAddress, address _advisorAddress,
   address _incentivisationFundAddress, address _enjinTeamAddress)
ERC20Token("Enjin Coin", "ENJ", 18)
 {
    crowdFundAddress = _crowdFundAddress;
    advisorAddress = _advisorAddress;
    enjinTeamAddress = _enjinTeamAddress;
    incentivisationFundAddress = _incentivisationFundAddress;
    balanceOf[_crowdFundAddress] = minCrowdsaleAllocation +
       maxPresaleSupply; // Total presale + crowdfund tokens
    balanceOf[_incentivisationFundAddress] = incentivisationAllocation;
             // 10% Allocated for Marketing and Incentivisation
    totalAllocated += incentivisationAllocation;
                                  // Add to total Allocated funds
}
```

Standard constructor, just assigns state varibles, and sets the balance of the crowdfund contract's address to equal the crowdsale allocation + presale supply. Adds incentivization allocation and token balances as well.

transfer

```
function transfer(address _to, uint256 _value) public returns (bool
   success) {
    if (isTransferAllowed() == true || msg.sender == crowdFundAddress ||
       msg.sender == incentivisationFundAddress) {
        assert(super.transfer(_to, _value));
        return true;
    }
    revert();
}
```

The transfer function wraps around ERC20's transfer to be able to enforce that only the incentivization and crowd fund addresses are able to transfer if transfers are not yet enabled.

transferFrom

```
function transferFrom(address _from, address _to, uint256 _value) public
    returns (bool success) {
    if (isTransferAllowed() == true || msg.sender == crowdFundAddress ||
        msg.sender == incentivisationFundAddress) {
        assert(super.transferFrom(_from, _to, _value));
        return true;
    }
    revert();
}
```

Like the transfer function above, this just acts as a wrapper around the ERC20 transfer from to ensure tokens cannot be transferred until transfers are enabled.

releaseEnjinTeamTokens

```
function releaseEnjinTeamTokens() safeTimelock ownerOnly returns(bool
    success) {
    require(totalAllocatedToTeam < enjinTeamAllocation);

    uint256 enjinTeamAlloc = enjinTeamAllocation / 1000;
    uint256 currentTranche = uint256(now - endTime) / 12 weeks;     // "
        months" after crowdsale end time (division floored)

    if(teamTranchesReleased < maxTeamTranches && currentTranche >
        teamTranchesReleased) {
        teamTranchesReleased++;
        transferTeamAllocation(safeMul(enjinTeamAlloc, 125));
        return true;
    }
    revert();
}
```

This function allocates tokens to the Ejin team in tranches, every 3 months a new tranche allocation is possible, until the total allocation is exhausted.

transferTeamAllocation

```
function transferTeamAllocation(uint256 _amount) ownerOnly internal {
    balanceOf[enjinTeamAddress] = safeAdd(balanceOf[enjinTeamAddress],
        _amount);
    Transfer(0x0, enjinTeamAddress, _amount);
    totalAllocated = safeAdd(totalAllocated, _amount);
    totalAllocatedToTeam = safeAdd(totalAllocatedToTeam, _amount);
}
```

This function allocates tokens to the Enjin team. It is only callable through releaseEnjinTeamTokens.

releaseAdvisorTokens

```
function releaseAdvisorTokens() advisorTimelock ownerOnly returns(bool
    success) {
    require(totalAllocatedToAdvisors == 0);
    balanceOf[advisorAddress] = safeAdd(balanceOf[advisorAddress],
        advisorsAllocation);
    totalAllocated = safeAdd(totalAllocated, advisorsAllocation);
    totalAllocatedToAdvisors = advisorsAllocation;
    Transfer(0x0, advisorAddress, advisorsAllocation);
    return true;
}
```

This function allocates tokens to advisors. It can only be called once, as after the totalAllocatedAdvisors variable will no longer be zero. It can only be called by the owner, 8 weeks after the crowdsale has ended.

retrieveUnsoldTokens

```
function retrieveUnsoldTokens() safeTimelock ownerOnly returns(bool
    success) {
    uint256 amountOfTokens = balanceOf[crowdFundAddress];
    balanceOf[crowdFundAddress] = 0;
    balanceOf[incentivisationFundAddress] = safeAdd(balanceOf[
        incentivisationFundAddress], amountOfTokens);
    totalAllocated = safeAdd(totalAllocated, amountOfTokens);
    Transfer(crowdFundAddress, incentivisationFundAddress, amountOfTokens)
        ;
    return true;
}
```

This function assigns any unsold tokens to the incentivization fund address. It can only be called by the owner 24 weeks after the crowdsale has ended.

addToAllocation

```
function addToAllocation(uint256 _amount) crowdfundOnly {
    totalAllocated = safeAdd(totalAllocated, _amount);
}
```

Simply adds the amount of tokens to totalAllocated using SafeMath. Can only be called by the crowdfund contract.

allowTransfers

```
function allowTransfers() ownerOnly {
    isReleasedToPublic = true;
}
```

This function is called to enable token transfers by the public. It can only be called by the token contract owner.

isTransferAllowed

```
function isTransferAllowed() internal constant returns(bool) {
    if (now > endTime || isReleasedToPublic == true) {
        return true;
    }
    return false;
}
```

This function is the check for the transfer and transferFrom wrappers, it allows transfers if the crowdsale is over, once allowTransfers() has been called.

Summary

The contracts are well constructed and appear to work as intended, I have found no issues or major risks in any of the functions or implementation that would pose a risk to the team or crowdsale participants.