oceanico.io

Project

SMART CONTRACTS EXPERTISE — RIGHT WAY TO SUCCESS
Project Smart Contract Audit

If you have any questions concerning smart contract design and audit, feel free to contact zoia@oceanico.io

Content

Description of the complex of procedures for auditing a smart contract

1. Primary architecture review

- Checking the architecture of the contract.
- Correctness of the code.
- Check for linearity, shortness and self-documentation.
- Static verification and code analysis for validity and the presence of syntactic errors.

2. Comparison of requirements and implementation

- Checking the code of the smart contract for compliance with the requirements of the customer code logic, writing algorithms, matching the initial constant values.
- Identification of potential vulnerabilities

3. Testing according to the requirements

- Control testing of a smart contract for compliance with specified customer requirements.
- Running tests of the properties of the smart contract in test net.

Terms of Reference for the creation of a smart contract

ICO Crowdfunding Details

Project Crowdsale Details:

Token Specs:

Name: Project

Trading symbol: COS

Total Tokens: 333,000

ICO Details:

Sale duration: 8 weeks

Qty to be sold: 200,000,000

Minimum contribution amount: 1 ETH

The ICO pricing and allocation of the token per tier is shown below in the table. The price of the token is intentionally fixed to the USD rather than fluctuating with the price of Ethereum. Instead of the price of the tokens fluctuating with the price of ethereum the amount of tokens fluctuates with Ethereum. The value of Ethereum dictates the amount of tokens that can be purchased.

Stage	Token Amount	Token Price (USD)	Percent-age Paid	Token Price (USD)	Tier Value (USD)	Tier Value (GBP)	ETH=$1050 (6/1/18)	Mini-mum ETH Reg	Sales Required
Pre Seed	5,000,000	$0.44	40.0%	$0.016	$80,000.00	£58,966.96	76.122	1.19	67,227
Tier 1	7,500,000	$0.44	50.0%	$0.020	$150,000.00	£110,563.05	142.728	1.19	126,050
Tier 2	37,500,000	$0.44	60.0%	$0.024	$900,000.00	£663,378.30	853.368	1.19	753,303
Tier 3	37,500,000	$0.44	70.0%	$0.028	$1,050,000,00	£773,941.35	999.096	1.19	882,353
Tier 4	37,500,000	$0.44	80.0%	$0.032	$1,200,000.00	£884,504.40	1,141.824	1.19	1,008,403
Tier 5	37,500,000	$0.44	90.0%	$0.036	$1,350,000.00	£995,067.45	1,284.552	1.19	1,134,454
Tier 6	37,500,000	$0.44	100%	$0.000	$1,500,000.00	£1,105,630.50	1,427.280	1.19	1,200,504
Total	**200,000,000**				**$6,230,000.00**	**£4,592,052.01**	**5,927.9699319663**	**1.19**	**5,235,294**

During the time of the ICO the tokens will not be tradable by participants, however, once the crowdsale is complete and the whitelist participants have been approved then the token trade function will be unpaused by the Founder and CEO.

The remaining tokens, if there are any, will be split 50%/50% with the participants that purchased tokens in the first tiers marked in orange above, PreSeed, Tier 1, and Tier 2 and the token reserve. If there are less than 20,000 tokens left they will all be moved to the reserve. Each contributor that participated during in those tiers will receive an equal amount of the remaining tokens. For example, if there are 100,000 remaining, and there are 100 contributors, regardless of how much and how many times they contributed, each of the 100 will get an extra 500 tokens. The tokens are first split in half, 50,000 will go the reserve, the other 50,000 gets split among the 100 users evenly. To receive the bonus tokens you will have to be approved and added to the whitelist.

In order to get the bonus you have to check to see if you have been in fact whitelisted and also were lucky enough to purchase the tokens during the bonus tiers. To check, use the “checkContributorStatus” function in etherscan. If you meet that criteria then you will be able to request the tokens by using the “claimBonusTokens” function.

To participate in the ICO each individual will have to complete the online whitelist registration along with make a deposit. The information collected in the whitelist qualification will be verified on an individual basis. If during that process of verifying the information sent, the team finds that the information does not meet the criteria of the whitelist, the participant will be rejected. If rejected, the participant will be able to withdraw their funds after the ICO is complete, NOT before. If the participant information meets the criteria of the whitelist they will be approved. In the event they are approved the tokens they requested will be sent to them.

Once the ICO has ended the project founder and CEO will use the “Finalize” function to transfer the pot of team tokens into the team holding wallet where they will be held frozen for one year. After that time the team will be able to call that smart contract and move their tokens into their wallet. The finalize “function” will also clean up any dust that remains in the smart contract and transfer it to the holding wallet.

List of audited files

Github of the project

https://github.com/Project/smartcontracts

The following 4 .sol files were the object of the audit:

- COSCrowdSale.sol
- COSTeamWallet.sol
- COSToken.sol
- Migrations.sol

Review of smart contract #1

Project Crowdsale contract review #1

https://github.com/Project/smartcontracts

Important:

1. Solidity 0.4.19 released. It is recommended to update the compiler and change the pragma.

2. It is recommended to use template contracts (such as ERC20, ERC20Basic, StandardToken, Ownable, SafeMath, BasicToken) from zeppelin-solidity https://github.com/OpenZeppelin/zeppelin-solidity

3. Project Crowdsale Details states that the duration of the ICO is 8 weeks, but the code specifies 60 days.

```
92        icoEndTime = now + 60 days;
```

4. In the calculateUnsoldICOTokens() function the remainingTokens is not reset to 0 before the calculation. This will lead to the fact that the ICO has never finished on the number of tokens sold.

5. If the purchase of the number of tokens is larger than the remaining number of tokens to complete the tier, then the saleTier[tier].tokensSold for the current (not the next tier) does not change its value, but should equal saleTier[tier]. tokensToBeSold. This leads to another error in the calculation of unsold tokens by the method of calculateUnsoldICOTokens().

6. There are no guarantees that with each call of buyTokens there will be a current price in the contract ether in dollars. Only if it is in some way guaranteed by the customer.

7. In the buyTokens method, when you try to buy tokens at least (>=), what is left before tier is completed, the number of tokens credited is incorrectly considered. The problem is in the line on the screen. Here mul(price) should be replaced by div(price).

```
149            buyTokensRemainingWei = (remainingWei.mul(price)).mul(TOKEN_DECIMALS);
```

8. When selling all the tokens of the first tier, the following sales become impossible. Transactions are simply not executed. This is due to the problem from the previous paragraph. Fixes the same as the previous problem.

```
Error: VM Exception while processing transaction: invalid opcode
```

9. There is a bug when buying a number of tokens more than the remaining buy until the end of the current tier. Tokens from the new tier are calculated at the price of the previous tier.

10. There is one more bug when buying tokens from several tiers with one transaction. The tier counter changes only by 1 when buying tokens at once 3+ tiers.

11. It's unclear why the moveTokensForSale method is needed. TK does not assume such a functional. In the TOR, the number of tokens that must be sold per tier is fixed for each tier.

12. Still on this method. The require (paused = true) statement will not work as expected. If you need to check that the contract is now paused (paused == true), then there must be an equals (==) operator, not assignments (=). If you want to set paused to true, then require is not needed.

```
227   /// @notice used to move tokens from the later tiers into the earlier tiers
228   /// contract must be paused to do the move
229   /// param tier from later tier to subtract the tokens from
230   /// param tier to add the tokens to
231   /// param how many tokens to take
232   function moveTokensForSale(uint8 _tierFrom, uint8 _tierTo, uint256 _tokens)
233     public
234     onlyOwner
235   {
236     require(paused = true);
237     require(_tierFrom > _tierTo);
238     require(_tokens <= ((saleTier[_tierFrom].tokensToBeSold).sub(saleTier[_tierFrom].tokensSold)));
239
240     saleTier[_tierFrom].tokensToBeSold.sub(_tokens);
241     saleTier[_tierTo].tokensToBeSold.add(_tokens);
242   }
```

13. By the method of distributeRemainingTokens: it is unreliable to pass _totalNumWhitelisters as a parameter from outside. Bonus tokens can be distributed incorrectly if the parameter is incorrectly calculated. It is better to consider this number inside the method, let it take more gas.

14. A typo in the name of the token. "Project".

```
9   string public constant name = "Project";
```

15. If the user reserved tokens, then the whitelistAddresses method with Denied status for the user was performed (for example, by mistake), then whitelistAddresses was already executed with the Accepted status for that user, then the user will not receive the tokens that he reserved before it was executed first whitelstAddresses. The fact is that the number of tokens of this user is reset to zero on the first call.

```
218   /// @notice allows denied buyers the ability to get their Ether back
219   function deniedWhitelistAddress(address _investorAddress)
220     internal
221   {
222     require(_investorAddress != 0x0);
223     investors[_investorAddress].whitelistStatus = Status.Denied;
224     investors[_investorAddress].qtyTokens = 0;
225   }
```

16. The COSToken.transferFrom () method does not translate tokens because allowed [_from] [_ to] is always zero and does not change anywhere, and the require(_value <= allowed [_from] [msg.sender]) test results in a reverse . This applies to all COSCrowdSale methods of the contract, where there is a translation of the tokens.

17. According to the Terms of Reference *"Once the ICO has ended the project, the founder and CEO will use the" Finalize "function to transfer the pot of the team tokens into the team holding the wallet where they will be frozen for one year",* in the contract code the countdown of 1 year goes from the moment of creation of the contract COSTeamWallet, and not from the moment of call Finalize.

18. *"If rejected, the participant will be able to withdraw their funds after the ICO is complete"*. In the code, there is no such possibility.

Code quality:

19. The pausedContract modifier actually checks that paused == false, it is not obvious and misleading. I recommend that you rename the modifier to notPausedContract.

```
76    modifier pausedContract(){
77      require(paused == false);
78      _;
79    }
```

20. getEtherPrice sets a new price, and does not return it. The same thing - is misleading. I recommend that you rename the function to setEtherPrice

```
181   function getEtherPrice(uint256 _price)
182     external
183     onlyOwner
184     {
185       ethPrice = _price;
186     }
```

21. In the COSCrowdSale contract, minLimit is declared, but not used. It is necessary to remove. Extra waste of gas.

```
24    uint256 public minLimit;
```

22. Indents. Somewhere 2 characters, somewhere 4, in some places tabulation, and somewhere in general there is no indentation. From this you need to get rid of. It is advisable to bring everything to 2 (4) whitespace indentations.

```
192   function whitelistAddresses(address[] _addresses, bool _status)
193     public
194     onlyOwner
195     {
196         for (uint256 i = 0; i < _addresses.length; i++) {
197             address investorAddress = _addresses[i];
198             if(_status == true){
199                 approvedWhitelistAddress(investorAddress);
200             } else {
201                 deniedWhitelistAddress(investorAddress);
202             }
203         }
204   }
```

```
14    function COSToken()
15      public
16    {
17      // 333,000,000 total supply of COS tokens
18      totalSupply = 333000000 * 10**10;
19      paused = true;
20
21
22        balances[msg.sender] = totalSupply;
23      Transfer(0, owner, totalSupply);
24
25      // making sure the msg.sender and the owner are the same, and that the
26          // address of the owner recieved the totalSupply of tokens.
27      assert(balances[owner] == totalSupply);
28    }
```

```
38    function activate()
39    public
40    onlyOwner {
41    paused = false;
42    }
```

23. In the checkContributorStatus() method, it is better not to use the number 3, which is explicitly specified. Use the constant BONUS_TIER instead.

```
262   function checkContributorStatus()
263     view
264     public
265     returns (bool whitelisted, bool bonusTier)
266   {
267     return (investors[msg.sender].whitelistStatus == Status.Approved, investors[msg.sender].tierPurchased < 3);
268   }
```

24. In the contract COSTeamWallet withdrawalAddress and totalWithdrawn are defined but not used.

Review of smart contract #2

Project Crowdsale contract review #2

https://github.com/Project/smartcontracts

Important:

1. COSTeamWallet. Added 60 days to FREEZE_TIME. There is no guarantee that the ICO will end in 60 days. It can end earlier if all tokens are sold earlier. Tokens should be frozen for 365 days from the time the finalize function is called. I recommend that the COSTeamWallet contract be added to the token freeze function, which will be called from COSCrowdSale in the finalize function.

2. In line 236, instead of < should be <= .

```
230   function checkContributorStatus()
231     view
232     public
233     returns (bool whitelisted, bool bonusTier)
234   {
235     Participant storage participant = participants[msg.sender];
236     return (participant.whitelistStatus == Status.Approved, participant.tierPurchased < BONUS_TIER);
237   }
```

3. The check requires(_address! = 0x0) from approveAddressForWhitelist and denyAddressForWhitelist. I recommend to return it.

4. You should pass a block.timestamp or now instead of a block.number to the setFreezeTime function and you should replace teamWallet in finalize with bnbTeamWallet and remove unused teamWallet.

Code quality:

In case the investor wants to make a refund, and there will not be enough funds on the contract account, then it will not be possible to make a refund. It is necessary to wait until the owner of the contract replenishes the account. I recommend to implement both in zeppelin-solidity and their RefundableCrowdsale.

Review of smart contract #2

Testing

Plan:

1. Make a purchase for the amount of <MIN_CONTRUBUTION. Verify that the transaction has not passed.

2. Make a simple purchase of several ethers within one Tier. Make sure that there are as many tokens credited as expected.

3. Make a purchase of the remaining tokens in the first Tier.

4. Make a purchase of tokens of all Tier 1 + several from Tier 2 from the second account. Make sure that the price of the token in the next Tier is calculated correctly and does not equal the price of the token in the previous Tier.

5. Make a purchase immediately for a few Tier from the second address. Make sure that you bought as many tokens as expected.

6. Verify that finalize, claimRefund, claimRemainingWei, and claimTokens are unavailable until the end of the ICO.

7. From the third account, buy the remaining tokens from all Tier + try to buy some more in the same transaction. Make sure that tokens are bought no more than possible.

8. Try to buy more tokens. Verify that the transaction failed.

9. Run approveAddressForWhitelist for the first and third addresses and denyAddressForWhitelist for the second, finalize, distributeRemainingTokens.

10. Make a refund at the denied address. Check that the refund does not work for approved addresses.

11. Make claimBonusTokens from the first and check that it does not work with the second and third.

12. Make claimTokens from the first and third addresses and check that it does not work with the second one.

13. Make claimRemainingWei from all addresses. Check that the funds are returned.

14. Check the balance of teamWallet tokens, make sure that there is an expected number of tokens.

Review of smart contract #2

The process:

1. **Success.** The price of 1 ETH is set at $ 40’000. I tried to buy tokens by 0.5 ETH. The transaction failed.

2. **Success.** Tokens are bought for 1 ETH. 2’500’000 tokens are reserved. All as expected.

3. **Success.** Completed purchase for another 1 ETH. A further 2’500’000 tokens are reserved.

4. **Success.** Sent 4 ETH. 7’916’666 tokens are reserved (7’500’000 from Tier 1 + 416’666 with Tier 2). That’s right.

5. **Success.** Everything is the same.

6. **Success.** Transactions have not passed.

7. **Success.** The price is set to 1 ETH = $ 80’000, so that the test account has enough funds to complete the ICO. The expected number of tokens is reserved.

8. **Success.** The transaction failed.

9. **Success.** approveAddressForWhitelist, denyAddressForWhitelist, finalize was completed. The distributeRemainingTokens transaction did not pass because the remainingTokens == 0.

10. **Success.** Refund to the denied address worked after the contract account is replenished. For approved addresses, the transaction failed.

11. **Success.** Transactions did not pass, because distributeRemainingTokens was not performed at point 9.

12. **Success.** claimTokens works for approved addresses. Tokens are accrued. For denied addresses, transactions do not pass.

13. **Success.** claimRemainingWei worked for approved addresses for which there were funds left.

14. **Success.** Everything is the same.

Review of smart contract #3

Project Crowdsale contract review #3

https://github.com/Project/smartcontracts

Testing

Plan:

1. Contracts deployment.
2. Initial preparing.
3. Buy tokens for less than MIN_CONTRIBUTION.
4. Simple purchase of several tokens in PreSeed. Check that you bought as many tokens as you expected.
5. Buy remaining tokens in PreSeed.
6. Buy all tokens from Tier1 and some tokens from Tier2 from second account.
7. Buy remaining tokens in Tier2, all tokens in Tier3 and some Tokens in Tier4 from second account.
8. Withdraw funds. Verify that all the funds of the contract were transferred to _ holdings address.
9. Attempt to execute finalize, claimRefund, claimRemainingWei, claimTokens before all tokens are sold.
10. Buy all remaining tokens and try to buy more.
11. Attempt to buy more tokens when all tokens are already sold.
12. Run approveAddressForWhitelist for the first and third addresses and denyAddressForWhitelist for the second, finalize, distributeRemainingTokens.
13. Make a refund from the denied address. Check that the refund does not work for approved addresses.
14. Execute claimBonusTokens from the first account and check that it does not work with the second and third.

15. Execute claimTokens from the first and third addresses and check that it does not work with the second one.

16. Execute claimRemainingWei from all addresses. Check that the funds are returned.

17. Check the balance of teamWallet tokens, make sure that there is an expected number of tokens.

Testing in testnet

1. **Success** Deployed COSToken, COSTeamWallet, COSCrowdSale contracts.

2. **Success** Token.approve, token.setCrowdsaleContract, wallet.setCrowdsaleContract. Set ether price to 40'000.

3. **Success** Tried to buy tokens at 0.5 ETH. Rejected.

4. **Success** Bought 2'500'000 tokens for 1 ETH.

5. **Success** Bought 2'500'000 tokens for 1 ETH.

6. **Success** Bought 7916666 tokens for 4 ETH.

7. **Success** Bought it for 40 ETH. Reserved 62'440'476 tokens.

8. **Success** Executed ownerWithdrawal. Checked balance of _holding and crowdsale.

9. **Success** Rejected.

10. **Success** Set ether price = 80'000. Buy tokens for 60 ETH from third account. Reserved 124'642'858. It's all remaining tokens.

11. **Success** Transaction rejected.

12. **Success** Transactions was successfully completed. The distributeRemainingTokens transaction did not pass because the remainingTokens == 0.

13. **Success** Refund to the denied address worked after the contract account is replenished. For approved addresses, the transaction failed.

Review of smart contract #3

14. **Success** Transactions did not pass, because distributeRemainingTokens was not performed at point 12.

15. **Success** claimTokens works for approved addresses. Tokens are accrued. For denied addresses, transactions do not pass.

16. **Success** claimRemainingWei worked for approved addresses for which there were funds left.

17. **Success** Everything is the same.

Results of contract audit

https://github.com/Project/smartcontracts

The information in this report is a list of tips and recommendations on what to look for and what needs to be done to ensure the performance of a smart contract.

The OCEANICO experts conducted the verification of the smart contract in three iterations and completed the full testing. Based on the results of each iteration, the customer's developers were given recommendations for optimizing the smart contract code to fix bugs and vulnerabilities.

This smart contract complies with the specifications specified in the terms of reference, full testing has shown and does not contain previously identified code and vulnerabilities errors.

If changes are made to the functionality of the contract, please submit the smart contract for re-examination to the OCEANICO experts (zoia@oceanico.io).