VUULR ICO Audit Report (April 17)

Summary

Vuulr intends to run a crowdsale commencing in 2018.

Bluzelle Network was commissioned to perform an audit on Vuulr's ico vesting and token Ethereum smart contract.

This audit has been conducted on Vuulr's source code in GitHub commit 65a954744977336d1e7f202fe59e637a4dc53b99.

Scope

All Solidity code contained in TGE-Smart-Contracts-master/contracts/ was considered in scope along with the test reports as a basis for static and dynamic analysis. Please note Bluzelle is not responsible for open-zepplin contracts and therefore did not test them on an individual basis.

Focus Areas

1. Correctness

- Adherence to adopted standards such as ERC20.
- Test executions to see generic incorrect behavior.

2. Testability

- Testing of all functions to make a high level sanity check they function as expected
- Validation that events in general are firing as expected
- Deployment onto local test environments to ensure deployment and construction of classes is functional
- Basic edge-case testing

3. Security

- Overview looking for known security weaknesses
- Looking to ensure ETH or tokens are not at risk of malicious attempts to withdraw/transfer
- Looking at integer overflow or underflow and proper use of math functions

4. Best practice

- Explicit labeling for the visibility of functions and state variables
- Enforcing max gas prices to prevent buyers from gaming the ICO

Classification

Defect Severity

- Minor - A defect that does not have a material impact on the contract execution and is likely to be subjective.
- Moderate - A defect that could impact the desired outcome of the contract execution in a specific scenario.
- Major - A defect that impacts the desired outcome of the contract execution or introduces a weakness that may be exploited.
- Critical - A defect that presents a significant security vulnerability or failure of the contract across a range of scenarios.

Findings

Minor

- **Update README.md file outlining deployment steps to increase transparency** - We recommend updating README.md file detailing deployment steps so contributors can best understand the procedure of participating.
- **Minor Issues listed for General, Ownable Contract, Operatable Contract, XClaimable Contract, Salvageable Contract, VUULRToken Contract and VUULRVesting Contract –** Please review the minor issues listed for these sections.

Moderate

- **Moderate Issue listed for VUULRVesting Contract –** Please review the moderate issue listed for this section.

Major

- None found.

Critical

- None found.

Testing

To further satisfy test coverage, the contracts were tested on a local network (using Ganache) and also on the Ropsten Test Network to achieve simulation of a mock sale.

Conclusion

We are pleased to report that no potential Major or Critical vulnerabilities were uncovered during this audit. The token contract complies with the general ERC20 Token Standards and could have a few more improvements to adhere to additional EIP-20 standards. The code has good testability.

Of the issues we have raised, all of them were minor issues, with the exception of one moderate issue. We feel that making the recommended changes will help the crowdsale become more secure and thus reduce the risk of ethereum being hacked or stolen.

General

Recommendations (Best Practices):

- (**Minor Issue**) There is a chance that when deploying the ico contracts or uploading data via registerVestingSchedule to the main ethereum network they will halt. If a js script is used, they may give an error similar to one shown below. This is because the transactions are not getting mined fast enough. To overcome this problem, please ensure to give above-average amount of gas.

  (node:7638) UnhandledPromiseRejectionWarning: Unhandled promise rejection (rejection id: 1): Error: Transaction was not mined within 50 blocks, please make sure your transaction was properly send. Be aware that it might still be mined!

Ownable, Operatable, OperatableBasic, Claimable, and XClaimable Contracts

Recommendations (Best Practices):

- (**Minor Issue**) Add **events** for initiating Ownership transfers and also canceling Ownership pending transfers. These events can be named as OwnershipTransferInitiated(pending_address) or OwnershipTransferCanceled() or something similar. Note that in the open-zepplin framework, an event is triggered for every Ownership Transfer.
- (**Minor Issue**) Consider having cancelOwnershipTransfer() set pendingOwner as address(0) like what claimOwnership(...) function does when a pending transfer is completed- though this means that it is mandatory for the function transferOwnership(...) to check for newOwner not to be equal to address(0).
- (**Minor Issue**) Add **events** for primaryOperator and secondaryOperator changes. For example in open-zepplin framework, an event is triggered for every Ownership Transfer. They can be named something along the name PrimaryOperatorpdated(address) or SecondaryOperatorUpdated(address).
- (**Minor Issue**) Add public view isOwner(address _address) similar to isPrimaryOperator(...) and isSecondaryOperator(...).
- (**Minor Issue**) If possible, have transferOwnership(address newOwner) modified to check if the inputted address is not address(this) (the contract's address). You may also have it check that the newOwner is not owner. As for setPrimaryOperator(address addr) and setSecondaryOperator(address addr) have it check that proposed address is not address(this).
- (**Minor Issue**) In the function transferOwnership(newOwner) have it modified to return a boolean. For example, in transferOwnership function you would have require lines at the top (require != address(0), address(this), ..etc.) and then at the end of the function return true. Likewise make the functions transferOwnership(...), cancelOwnershipTransfer(...) and claimOwnership(...) return true. Similarly have the functions setPrimaryOperator(address addr) and setSecondaryOperator(address addr) return booleans (ie. return true upon success).

Salvageable Contract

Recommendations (Best Practices):

- (**Minor Issue**) Add an event EmergencyERC20Drain(address tokenaddress, uint256 _amount) whenever emergencyERC20Drain(...) is called.

- (**Minor Issue**) The function `emergencyERC20Drain(ERC20 oddToken, uint amount)` should return a boolean. For example, if amount = 0, the function can return `false` and if not then it'll perform the transfer (issue an event in accordance to previous issue), and then return `true`.

VUULRToken Contract

Recommendations (Best Practices):

- (**Minor Issue**) Consider adhering to optional EIP 20 standards
  https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
  Specifically, adding public view getter functions such as name(), symbol() and decimals().
  See .sol examples presented in the link above.

bluzelle

VUULRVesting Contract

Questions:

- **(Minor Issue)** Because the function whichPeriod(...) is declared as public view and not used in any other function, we're assuming it is intended for public use– specifically, for vestees. However, the additional check for started boolean on line 104 may yield confusing results to an early vestee. Because if vesting has not started, no matter what time value a vestee enters into the function, they will always get 0- thinking that they were not registered into the system correctly. We'll leave it to the Vuulr team to decide if this issue on line 104 is necessary to change – provided that it doesn't cause major problems in other parts of the code or other contracts– such as the crowdsale contract.
- **(Minor Issue)** What strategies are in place if large amounts of tokens were not withdrawn by inactive vestees in the far future (5 years from now for example). Would Vuulr team allow it to sit with VUULRVesting contract forever? Keep in mind, if it is planned to call revokeSchedule(...), then most likely all tokens will be treated as withdrawable and therefore get sent to the revoke_address which has been inactive. If changeVestingAddress(...) is going to be use, then a new ethereum address has to be created for every inactive vestee. Also determining all inactive vestee by itself is a very burdensome task- assuming that there is no automated script that cross checks all withdrawing events with a local vestee database.

Recommendations (Best Practices):

- **(Moderate Issue)** We recommend that Vuulr Team to reserve a few tokens for situations when they have to compensate revoked vestee who receive less tokens then they're promised due to round-off errors. This will also ensure to resolve any complaints made peacefully.

  As an example, we added a vestee with a start time of October 20 2018 (1539993600), for 1,500,000 Tokens spanning 18 stages for 3 months (90 days) lock period.

  https://ropsten.etherscan.io/tx/0x73ed560eb35b93eac2a9c8f8abb907b1fbd000e41f206ae6e8a4a3835867c6a3

  Then sometime around the half way mark, during December 20 2020 (1608422400) at stage 9 (out of 18) the vestee withdraws tokens. The total amount this vestee receives at this point of time is 749,999.999999999999999997 VUULR Tokens, which is 3 base tokens less then what's expected. This is clearly due to round-off errors, since 18 doesn't divide evenly into 1.5 M.

  https://ropsten.etherscan.io/tx/0xcb6c659ba956d56f9178f26f9a9588c90255de013c3225c59672a06f37ec9418

A revoked vestee may complain about the 3 base tokens they didn't receive and even make claims that the smart contract has a bug that's "nickeling and diming" stakeholders. It's important to have a number of tokens reserved to handle these complaints peacefully by compensating the difference – no matter how minuscule the amount is.

Having all potential vestees fully understand how calculations could be off during the intermediate periods of their withdraw schedule is a good way to prevent future conflicts. At the same time, Vestees need to be ensured that when the scheduling period ends, they will indeed receive the full amount of tokens reserved for them.

One strategy to prevent round off errors is to ensure that the value assigned for number of periods/stages is a number that can divide evenly into the number of tokens. For example, if 1.5 M is going to be assigned to a vestee, then having the number of stages set as 15 would be a better solution. This would ensure that at the half way mark (stage 8), they would receive exactly 800 K should they be revoked from the investment at that period of time.

- **(Minor Issue)** We recommend that Vuulr Team to have a contingency plan in place for situations where vestee's make complaints about receiving less tokens during an intermediate point of time during their withdraw schedule due to round-off errors.

  As an example, we added a vestee with a start time of September 2018 (1536624000), for 800,000 Tokens spanning 12 stages for 1 month (30 days) lock period.

  https://ropsten.etherscan.io/tx/0x526aeb2520dc466659a260267bb02c94ca13a4975cb7e4e6787fbca6b7c81295

  Then sometime around the half way mark, during March 2019 (1551916800) at stage 6 (out of 12) the vestee withdraws tokens. The total amount received at this point of time is 399,999.999999999999999996 VUULR Tokens, which is 4 base tokens less then what's expected. This is clearly due to round-off errors, since 12 doesn't divide evenly into 800 K.

  https://ropsten.etherscan.io/tx/0x526aeb2520dc466659a260267bb02c94ca13a4975cb7e4e6787fbca6b7c81295

  Although very unlikely, this vestee may complain about the 4 base tokens they didn't receive. They may even make claims that smart contract has a bug that's "nickeling and diming" all stakeholders. It's important to be able to handle these complaints peacefully and ensure they fully understand why calculations could be off during the intermediate periods of the schedule. Furthermore, vestees need to be ensured that when the scheduling period ends, they will indeed receive the full amount of tokens reserved for them. In the example above, the amount they'll receive at the end is exactly 800 000 VUULR Tokens.

  One strategy to prevent round-off errors, is to ensure that number of periods/stages is a value that can divide evenly into the number of tokens. For example, if 800K is going to be assigned to a vestee, then having the number of stages set as 8 would be a better solution. This would ensure that at the half way mark (stage 4), they can withdraw exactly 400 K.

- (**Minor Issue**) Consider adding the following function:

  ```
  function currentTime() public constant returns (uint256) {
   return now;
  }
  ```

  and then using this function in place of all locations where the variable now is used. Example line 113:

  ```
  function vested(address beneficiary) public view returns (uint _amountVested) {

          ...
  if ((_vestingSchedule.tokens == 0) || (_vestingSchedule.numPeriods == 0) ||
  (currentTime() < _vestingSchedule.startTime)){

          ...
  ```

- (**Minor Issue**) If the function revokeSchedule(...) is going to be called frequently, consider simplifying it more or making it smarter. Currently it can be quite expensive to execute due to the two potential external calls to the vestingToken contract. For example, if _addressToRefund on line 160 will be the same for all cases whenever revokeSchedule(...) is called, then consider having a tab that sums up all the amounts that are _refundable. Then using this tab variable, have a separate function that would transfer all refundable tokens as a batch to the _addressToRefund function.
- (**Minor Issue**) Consider having the function start() return true when it succeeds.
- (**Minor Issue**) Consider having the function registerVestingSchedule(...) return true when it succeeds.
- (**Minor Issue**) Consider having the function withdrawVestedTokens() return true when it succeeds.
- (**Minor Issue**) Consider having the function revokeSchedule(...) return true when it succeeds.
- (**Minor Issue**) Consider having the function changeVestingAddress(...) return true when it succeeds.
- (**Minor Issue**) Consider having the function emergencyERC20Drain(...) return a boolean when it is executed.