**Introduction**

In this Smart Contract audit we’ll cover the following topics:

1. Disclaimer
2. Overview of the audit and nice features
3. Attack made to the contract
4. Critical vulnerabilities found in the contract
5. Medium vulnerabilities found in the contract
6. Low severity vulnerabilities found
7. Good to have
8. Summary of the audit

**1. Disclaimer**

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

**2. Overview**

The project has only one file, the VTOSToken.sol file which contains 212 lines of Solidity code. All the functions and state variables are well commented using the natspec documentation for the functions which is good to understand quickly how everything is supposed to work.  
*Nice Features:*  
The contract provides a good suite of functionality that will be useful for the entire contract: It uses [SafeMath](https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol) library to check for overflows and underflows which is a pretty good practise.

**3. Attacks made to the contract**

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

* **Over and under flows** An overflow happens when the limit of the type variable uint256 , 2 \*\* 256, is exceeded. What happens is that the value resets to zero instead of incrementing more.

For instance, if I want to assign a value to a uint bigger than 2 \*\* 256 it will simple go to 0 — this is dangerous.

On the other hand, an underflow happens when you try to subtract 0 minus a number bigger than 0.For example, if you subtract 0 - 1 the result will be = 2 \*\* 256 instead of -1.

This is quite dangerous. However This contract checks for overflows and underflows in [**OpenZeppelin's** *SafeMath*](https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol) and there is no instance of direct arithmetic operations.

* ***Short address attack*** This attack affects ERC20 tokens, was discovered by the Golem team and consists of the following:

A user creates an ethereum wallet with a trailing 0, which is not hard because it’s only a digit. For instance: 0xiofa8d97756as7df5sd8f75g8675ds8gsdg0

Then he buys tokens by removing the last zero:  
Buy 1000 tokens from account 0xiofa8d97756as7df5sd8f75g8675ds8gsdg

If the token contract has enough amount of tokens and the buy function doesn’t check the length of the address of the sender, the Ethereum’s virtual machine will just add zeros to the transaction until the address is complete.

The virtual machine will return 256000 for each 1000 tokens bought. This is a bug of the virtual machine that’s yet not fixed so whenever you want to buy tokens make sure to check the length of the address.

The contract isn’t vulnerable to this attack since it doesn't have any Buy function but also it **does NOTHING to prevent**the *short address attack* during **ICO** or in an **exchange** (*it will just depend if the ICO contract or exchange server checks the length of data, if they don't, short address attacks would drain out this coin from the exchange*). You can read more about the attack here: [ERC20 Short Address Attacks](http://vessenes.com/the-erc20-short-address-attack-explained/)

**4. Critical vulnerabilities found in the contract**

* As per ERC20 standard transfer, transferFrom and approve methods returns boolean indicating whether the said operation is successfully completed or not. This is critical as it makes the token non ERC20 compliant.
* Lines from line number 100 to 109 are commented which makes tokenSale method literally do nothing. Either the code needs to un-commented or remove the tokenSale method.

Generally the commented code should not be the part of deployable code.

* Because of above problem the fallback method at line 90 does nothing since it calls tokenSale. So the investor who will pay to this contract will receive nothing in return and their money will be lost.
* sendVTOSToken- In line 127 we check whether \_tokenLeft>=value whereas in line 130 we deduct the value from the owner's balance. Since the synchronicity between the \_tokenLeft and balance of owner is not maintained anywhere, there will be a case when \_tokenLeft>=value but balances[owner]<value.
* Since this contract supports payable(fallback) method and hence receiving ethers. There should be a method using which owner or other parties are able to withdraw those ethers from the contract, otherwise those funds will always be kept with the contract and no-one will be able to access those funds. Since contract accounts are not external accounts hence withdraw method should be there for thos contracts which are collecting funds in form of ether or any other tokens.

In line 127 instead of (\_tokenLeft>=value) please check (balances[owner]>=value);

* sendVTOSTokenToMultiAddr- The conditions is not checked whether the owner has the required amount of the token available with it before adding it to the receivers address. This will create invalid state and receivers will receive tokens more than the supplied tokens.

Though the sub method of safeMath will not allow you to minus something from 0 but still code should also handle this vulnerability.

Please add

require(balances[owner]>=amount[i]); after line 139

* destroyVTOSToken- Since you are removing tokens from the 'to' address and not assigning to anyone, the tokens are lost or burned and hence the totalSupply should also be reduced otherwise it will be an inconsistent state where total of all balances will be less than totalSupply, It will be good if you also keep track of the burned tokens so far with variable like \_totalBurned and update it everytime tokens are burned or destroyed.

Following lines needs to be added to maintain the consistent state

\_totalSupply=\_totalSupply.sub(value) after line 151

* destroyVTOSToken- In line 149 we check \_totalSupply>=value whereas in line 151 we reduce the amount from the 'to' balance. Instead we should check whether the said 'to' account actually contains tokens>=value. Replace line 149 with

require(to != 0x0 && value > 0 && balances[to] >= value);

**5. Medium vulnerabilities found in the contract**

* You’re specifying a pragma version with the caret symbol (^) up front which tells the compiler to use any version of solidity bigger than 0.4.11 .

This is not a good practice since there could be major changes between versions that would make your code unstable. That’s why I recommend to set a fixed version without the caret like 0.4.11.

* I would recommend you to use latest versions of solidity compiler instead of older versions as latest versions contains many critical bug fixes. Using compiler version 0.4.19 is recommended.
* The constructor of the ERC20 token should not payable. Please remove payable from line #82

**6. Low severity vulnerabilities found**

* Use view or pure instead of constant- This will increase the readability of the functions and will more clearly specify the purpose of the function. view functions means that they will not make any changes to the storage but will make reads from the storage.

Pure functions means they will neither make any changes to the storage nor will read from the storage.

Ex change function function mul() of SafeMath to

function mul(uint256 a, uint256 b) internal pure returns (uint256) {

uint256 c = a \* b;

assert(a == 0 || c / a == b);

return c;

}

* Token name and symbol should not be same. Token name identifies the proper name of the token where symbol is like identifier. For example name= GOLD PHOENIX, symbol- GPHX
* Comment at line #63 should be owner of the contract. Since specified owner is not the owner of the tokens but the contracts. Owner of the tokens are the token holders.
* As per ERC20 token standard decimals should be unit8 instead of uint256. If you mention uint it is by default uint256.
* Assign all functions the visibility modifier like public, internal or external. By default the modifier is assumed to be public by the solidity compiler.
* \_tokenLeft should be removed as its purpose is not clear.
* After line 85 a Transfer event should be fired since you are actually assigning/transferring all the tokens to the owner.

**7. Good to have**

* In approve and transferFrom please keep note of <https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729>
* It will be good to have if in approve you check the sender has the balance before approving and has not approved others to use that balance.

Ex- Alice has 1000 tokens and she has already approved bob to spend 500 tokens on behalf of her now she wants to allow Sameep to spend 600 tokens on her behalf. So if we will look Alice has given capacity to spend 1100 tokens on her behalf whereas she only owns 1000 tokens. So when she was allowing sameep to spend 600 on her behalf the transaction should be reverted. This is good to have.

**8. Summary of the audit**

* Overall the code is well commented. My final recommendation would be to pay more attention to the visibility of the functions since it’s quite important to define who’s supposed to executed the functions and to follow best practices regarding the use of assert, require etc. (which you are doing ;). I will also recommend to not put commented code. Because of commented code certain portion of the contract is useless and hence the contract is not ready for deployment and is not safe to use, since the commented code deals with the payable methods and will result in the loss of funds.
* All the ERC20 functions are included but approve, transfer and transferFrom functions are not ERC20 compliant.