**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. 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 ParsecPresale.sol file which contains 335 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.

**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. This contract checks for overflows and underflows but it will be good if the contract will use- [link](https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol). There are certain instances where multiplication and division are not properly checked.

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

* Constructor at line number 97 is payable and contains empty block. Please either put in some code which needs to be executed or remove the constructor. A method should only be payable if it expects to receive ether.
* addToWhiteList method defined at line number 293 is private and is not used anywhere. Because of this nobody will be able to buy tokens in first 2 hours of the sale, since first 2 hours of sale only allows whitelisted investor to put in his/her money.

Either use this method or make this method public and usable by owner only, so that owner may be able to add senders to white list.

**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 0.4.17 .

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.
* At line number 117, 120, 126, 164, 177, 210, 211, 233, 256 - Avoid to make time-based decisions in your business logic. The timestamp of the block can be manipulated by the miner, and all direct and indirect uses of the timestamp should be considered.Block numbers and average block time can be used to estimate time.
* At line number 187- Call to external contract should be done at the last after making changes to the state variables. This might lead to re-enterancy problem- [link](https://consensys.github.io/smart-contract-best-practices/recommendations/#external-calls)

Change

parsecToken.transfer(owner, unspentAmount);

unspentCreditsWithdrawn = true;

to

unspentCreditsWithdrawn = true;

parsecToken.transfer(owner, unspentAmount);

* At line number 202- Follow previous suggestion made.
* At line number 202- unspend amount is calculated after deducting grantedParsecCredits from the token balance of the contract address. Suppose owner calls this method after the token withdrawal has started and some investors has already claimed their credit tokens. In this case the balanceOf the contract will reduce and their might be the case when parsecToken.balanceOf(this) is less than grantedParsecCredits. Though this will be catched in safeDecrement, still it should be specifically handled to remove any confusion. Or an extra check should be there to check if token withdrawal has started or not.
* At line number 202- Call to external call is made and after that state variables are changed at line 223 and 226. This may lead to re-entrancy as discussed above. Please change it to following code-:

var tokensToSend = creditBalanceOf[msg.sender];

// Update amount of Parsec credits spent

spentParsecCredits = safeIncrement(spentParsecCredits, creditBalanceOf[msg.sender]);

// Participant's Parsec credit balance is reduced to zero

creditBalanceOf[msg.sender] = 0;

// Give allowance for participant to withdraw certain amount of Parsec credits

parsecToken.approve(msg.sender, tokensToSend);

* At line number 276- Follow previous suggestions.

**6. Low severity vulnerabilities found**

* At line number 307- Check for overflow and underflow is missing while multiplying and dividing 2 numbers. I would suggest you to use open-zepplin's SafeMath.sol for proper overflow and underflow checking.
* Please fix all the FIX ME comments.
* Fallback method will consume more than 23000 gas units. So if someone will send ether from the contract using send() or transfer() methods, it will fail. Consider this also.
* Certain parameters in the code are hard coded, like token address. It will be good is such parameters are passed as paramters while deploying contract.

**7. Summary of the audit**

* Overall the code is well commented.
* My final recommendation would be to pay more attention to the external functions which are called from the contract.
* 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.
* The contract has 'Speed Bumps' which allows for sometime to recover in case of attacks.
* The contract checks for the modifiers very proficiently.
* The use modifiers in the functions and state variables are explicitly specified which increases the readability of the contract and makes it more trustworthy.
* It will be good if contract will have a circuit breaker to stop contract execution in case of any emergency or attacks- [link](https://consensys.github.io/smart-contract-best-practices/software_engineering/#circuit-breakers-pause-contract-functionality).