SESAME SMART CONTRACT AUDIT

January, 2019

Block Consilium

Introduction

Sesame asked us to review their upcoming token contracts. *Block Consilium* reviewed the system from a technical perspective looking for bugs in their codebase. Overall the code is very well written. Read more below.

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

1. Disclaimer

The audit makes no statements or warrantees 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 one Solidity file for the Sesame ERC20 Token Smart Contract, the Sesame.sol file which contains 361 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 AND It **USES** SafeMath library to check for overflows and underflows which protects against underflow and overflow attacks, All the ERC20 functions are included it's a valid ERC20 token and in addition has some extra functionality for Burning, Selling Tokens and Setting Price.

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 **DOES** check for overflows and underflows by using **OpenZeppelin's** *SafeMath*, and there are no instances of direct arithmetic operations which might be dangerous. So this contract is **NOT vulnerable** to Overflow and Underflow bugs.

- **Replay attack** The replay attack consists of making a transaction on one blockchain like the original Ethereum's blockchain and then repeating it on another blockchain like the Ethereum's classic blockchain. The ether is transferred like a normal transaction from a blockchain to another. Though it's no longer a problem because since the version 1.5.3 of Geth and 1.4.4 of Parity both implement the attack protection EIP 155 by Vitalik Buterin
  So the people that will use the contract depend on their own ability to be updated with those programs to keep themselves secure.

- ***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 zeroes 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.

  Here is a **fix for short address attacks**

```
modifier onlyPayloadSize(uint size) {
    assert(msg.data.length >= size + 4);
    _;
}
function transfer(address _to, uint256 _value) onlyPayloadSize(2 * 32) {
    // do stuff
}
```

This contract **does not** implement an `onlyPayloadSize(uint numwords)` modifier for `transfer`, `transferFrom`, `approve`, `increaseApproval`, and `decreaseApproval` functions, it probably assumes that checks for short address attacks are handled at a higher layer (which generally are), and since the `onlyPayloadSize()` modifier started causing some bugs restricting the flexibility of the smart contracts, it's alright not to check for short address attacks at the Token Contract level to allow for some more flexibility for dAPP coding, BUT THE CHECKS FOR SHORT ADDRESS ATTACKS MUST BE DONE AT SOME LAYER OF CODING (e.g. for buys and sells, the exchange can do it - almost all well-known exchanges check for short address attacks after the Golem Team discovered it), this contract **DOES NOTHING to prevent the attack**, (probably in order to allow for greater flexibility and to prevent the errors caused by `onlyPayloadSize(uint numwords)` modifier) so the *checks for short address attcks must be done while buying or selling or coding a DAPP using SST*.

You can read more about the attack here: ERC20 Short Address Attacks.

- **Approval Doublespend**
  Imagine that Parul approves Arun to spend 100 tokens. Later, Parul decides to approve Arun to spend 150 tokens instead. If Arun is monitoring pending transactions, then when he sees Parul's new approval he can attempt to quickly spend 100 tokens, racing to get his transaction mined in before Parul's new approval arrives. If his transaction beats Parul's, then he can spend another 150 tokens after Parul's transaction goes through.

  This issue is a consequence of the ERC20 standard, which specifies that `approve()` takes a replacement value but no prior value. Preventing the attack while complying with ERC20 involves some compromise: users should set the approval to zero, make sure Arun hasn't snuck in a spend, then set the new value. In general, this sort of attack is possible with functions which do not encode enough prior state; in this case Parul's baseline belief of Arun's outstanding spent token balance from the Arun allowance.

It's possible for `approve()` to enforce this behavior without API changes in the ERC20 specification:
`if ((_value != 0) && (approved[msg.sender][_spender] != 0)) return false;`

However, this is just an attempt to modify user behaviour. If the user does attempt to change from one non-zero value to another, then the double spend can still happen, since the attacker will set the value to zero.

If desired, a nonstandard function can be added to minimize hassle for users. The issue can be fixed with minimal inconvenience by taking a change value rather than a replacement value:

```
function increaseApproval (address _spender, uint256 _addedValue)
onlyPayloadSize(2)
returns (bool success) {
  uint oldValue = approved[msg.sender][_spender];
  approved[msg.sender][_spender] = safeAdd(oldValue, _addedValue);
  return true;
}
```

Even if this function is added, it's important to keep the original for compatibility with the ERC20 specification.

Likely impact of this bug is low for most situations. This contract implements an `increaseApproval` and a `decreaseApproval` function, both of which takes the change in value instead of taking the new value, which is really *nice*.

For more, see this discussion on github: https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729

4. Critical vulnerabilities found in the contract

There aren't critical issues in the smart contract audited.

5. Medium vulnerabilities found in the contract

There are no Medium vulnerabilities in the contract.

6. Low severity vulnerabilities found

There are no low severity vulnerabilities in the contract.

7. Line by line comments

- Line 1:
  You're specifying a pragma version with the caret symbol (^) up front which tells the compiler to use any version of solidity bigger than 0.4.23.
  That's completely fine when you deploy the contract as it's already compiled, however to keep the code unbroken by future compilers just in case some backwards incompatible change is made, it is recommended to save the code without the caret symbol (^) for future use so that it forces to compile only with the version which was the latest or most stable when the code was written.

- Lines 3 to 31:
  SafeMath library is included for safe arithmetic operations.

- Lines 34 to 73:
  The `Ownable` contract makes the contract creator the owner of the contract, so that in `ERC20CrowdsaleToken` contract the contract creator becomes the owner and receives the initially minted tokens, and the owner is able to set price of tokens in terms of ether in the contract if needed.

- Lines 75 to 224:
  These lines implement the Standard ERC20 token features, along with `increaseApproval` and `decreaseApproval`functions to prevent `Approval Doublespend` attack.

- Lines 227 to 251:
  It implements a `BurnableToken` contract which means the token holders can `burn` tokens from their balance and thus decrease the total supply of the token, in general when the supply goes down and demand is either same or increases, the value of the tokens increases rapidly.

- Lines 253 to 268:
  The `Token` contract concludes the Burnable and Ownable ERC20 token part, it assigns the name `Sesame Token`, the symbol `SST`, decimals `18`, and initialSupply `10000000000` (10 Billion), to the token, and the `constructor` sends all the minted initial supply to the contract creator (`owner`).

- Lines 270 to 361:
  The `ERC20CrowdsaleToken` contract implements functionality required for crowdsale, it is implemented in such a way that the `owner` can start and stop sale (`startSale()` and `stopSale()` functions) anytime and can set the price of tokens in terms of ether anytime, (`setPrice()`), the buyer of the token will get the tokens immediately after sending ether to the contract if the contract has some tokens, if the contract does not have tokens for sale then the transaction

will fail. The incoming ether is forwarded to `owner` immediately in a successful transaction.

- Other nice features:
  The code uses new constructor syntax using the `constructor` keywords instead of same function name as contract's and there are no instances of the deprecated `throw` keyword, it uses new `assert(condition)`, `revert()`, `require(condition)` functions which is a nice thing to save gas. This contract is using the new syntax for emitting events with the `emit` prefix, which is a good thing, the new syntax for emitting events is the `emit` keyword followed by the event.

8. Summary of the audit

This code is very clean, thoughtfully written and in general well architected. The code conforms closely to the documentation and specification – we loved reading it.

The code is based on OpenZeppelin in many cases. In general, OpenZeppelin's codebase is good, and this is a relatively safe start.

Overall the code is well commented and clear on what it's supposed to do for each function. The visibility and state mutability of all the functions are clearly specified, there are no confusions. This is a secure contract that *will work as expected after reading the code*.