TWOGAP AUDIT

May, 2019

BLOCKCHAIN CONSILIUM

Introduction

TWOGAP asked us to review their upcoming token contracts. Blockchain Consilium reviewed the system from a technical perspective looking for bugs in their code base. 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 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 one Solidity file for the TWOGAP ERC20 Token Smart Contract, the TwogapContract.sol file which contains around 413 lines of Solidity code. We manually reviewed each line of code in the smart contract. 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 overflow and underflow attacks. All the ERC20 functions are included, it's a valid ERC20 token and in addition has some extra functionality for handling Bounties.

3. Attacks considered while auditing

In order to check for the security of the contract, we reviewed each line of code in the smart contract considering several known Smart Contract Attacks.

- **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 we 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.

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
}
```

Whether or not it is appropriate for token contracts to mitigate the short-address attack is a contentious issue among smart-contract developers. Many, including those behind the OpenZeppelin project, have explicitly chosen not to do so. Blockchain Consilium doesn't consider short address attack an issue of the smart contract at the token level.

This contract **does not** implement an `onlyPayloadSize(uint numwords)` modifier for `transfer`, `transferFrom`, `approve`, `increaseAllowance`, and `decreaseAllowance` 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 not prevent short address attack*, so the *checks for short address attack must be done while buying or selling or coding a DAPP using TWOGAP where necessary*.

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

- **Approval Double-spend:**
  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 behaviour 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 increaseAllowance (address _spender, uint256 _addedValue)
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 `increaseAllowance` and a `decreaseAllowance` 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#issuecomment263524729

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 5:
  The compiler version is specified as ^0.5.0, this means the code can be compiled with solidity compilers with versions greater than or equal to 0.5.0, that's totally fine, though it is recommended to remove the caret symbol while saving the code so that there are no unexpected issues with compiling in future when the compiler versions will be different.

- Lines 7 to 69:
  SafeMath library is included for safe arithmetic operations.

- Lines 71 to 130:
  The ERC20 Standard Interface is included, and a basic implementation of the ERC20 Standard is included.

- Lines 132 to 201:
  The `Ownable` contract makes the contract creator the owner of the contract, so that in `TwogapContract` the contract creator becomes the owner and receives the initially minted tokens, the `Ownable` contract has following noteworthy functions:

  1. `transferOwnership` allows the current owner transfer the control of the contract to a new Ethereum address when needed.

  2. `renounceOwnership` allows the current owner to relinquish control of the contract, if executed, it will not be possible to call owner-only functions anymore, for example, in this smart contract, if the owner renounces the ownership, they can no longer call the `bountyWithdraw` or the `withdraw` functions.

- Lines 203 to 400:
  The `TwogapERC20` contract implements the ERC20 standard interface and the Ownable contract.

  1. On line 207 a dictionary or `mapping` is declared to store the user's bounty balances.

  2. On line 209, `bountyLockTime3Month` refers to 26 Apr 2019, 12 pm UTC
  3. On line 210, `bountyLockTime6Month` refers to 26 Jul 2019, 12 pm UTC
  4. On line 211, `bountyLockTime12Month` refers to 26 Jan 2020, 12 pm UTC

5. From lines 219 to 232, the `bountyWithdraw` function along with the `withdraw` function allows the owner to transfer tokens from their account to another account as `bounty`.

6. From lines 234 to 336, standard ERC20 functions are included along with non-standard `increaseAllowance` and `decreaseAllowance` functions to account for approval double-spend attack.

7. From lines 338 to 356, the internal `_transfer` function is overwritten to handle bounties in such a way that users cannot transfer bounties before `bountyLockTime3Month`, cannot transfer more than 10% of their bounty tokens before `bountyLockTime6Month`, and cannot transfer more than 40% of their bounty tokens before `bountyLockTime12Month`.

8. From lines 358 to 371, an internal `mint` function is included to mint initial tokens and send them all to owner on contract creation.

9. From lines 373 to 399, internal `burn` and `burnFrom` functions are included, which are NOT used anywhere in this smart contract, *since they're not being used, it is recommended to remove them*.

- Lines 402 to 413.
  The `TwogapContract` implements the above `TwogapERC20`, and `ERC20Detailed` contracts, assigns the name "Twogap Token", the symbol "TGT", a supply of 210 Billion TGT, and 18 decimals. The constructor sends all initially minted tokens to owner.

- 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.

8. Improvement Suggestions

We suggest adding three more optional features.

- *Ability to Reclaim ERC20 Tokens transferred to wrong contract*

  If someone inadvertently sends ERC20 Tokens to a wrong contract address, and the contract does not have any mechanism to call `transfer` on an arbitrary token contract, the smart contract will never be able to claim any tokens held by itself and those tokens will be locked up in the smart contract forever.

  This can be avoided by adding a non-standard function to call `transfer` on an arbitrary token contract. That way, if someone inadvertently sends any ERC20 Tokens to this Token Contract, the owner will be able to call this function to transfer those tokens out of the contract.

  This can be done by adding the following (or similar) function to the `TwogapContract` contract.

```
function transferAnyERC20(address _tokenAddress, address _to, uint _amount)
    public onlyOwner {
      IERC20(_tokenAddress).transfer(_to, _amount);
  }
```

- *One step transfer to contract*

  Normally in ERC20 standard, when integrating the token with smart contracts or DAPPs for accepting payments, two steps are needed:

  1. The user approves the smart contract to transfer some amount their tokens on the user's behalf, using `approve` function.

  2. The smart contract then transfers the required tokens from the user's account to the beneficiary account and processes the payment.

  This way, normally two transactions are required and it spends more gas and more time than one transaction. It is recommended to include a one-step transfer to contract function to do the above process in one transaction and save time and gas fee.

  One step send (`approveAndCall`) can be implemented as follows:

```
interface tokenRecipient {
    function receiveApproval(address _from, uint256 _value, bytes calldata
_extraData) external;
}

contract TwogapContract is TwogapERC20, ERC20Detailed {

    uint8 public constant DECIMALS = 18;
    uint256 public constant INITIAL_SUPPLY = 210 * 10**9 * (10 **
uint256(DECIMALS));

    /**
     * @dev Constructor that gives msg.sender all of existing tokens.
     */
    constructor () public ERC20Detailed("Twogap Token", "TGT", DECIMALS) {
        _mint(msg.sender, INITIAL_SUPPLY);
    }

    function approveAndCall(address _spender, uint256 _value, bytes
calldata _extraData)
        external
        returns (bool success)
    {
        tokenRecipient spender = tokenRecipient(_spender);
        if (approve(_spender, _value)) {
            spender.receiveApproval(msg.sender, _value, _extraData);
            return true;
        }
    }
}
```

- *Two phase ownership transfer*

  This contract has a single owner, that owner can unilaterally transfer ownership to a different address. However, if the owner of the contract makes a mistake in entering the address of an intended new owner, then the contract can become irrecoverably unowned.

  In order to preclude this, Blockchain Consilium recommends implementing a two phase ownership transfer. In this model, the original owner designates a new owner, but does not actually transfer ownership. The new owner then accepts the ownership and completes the transfer.

  This can be implemented as follows:

```
contract Ownable {
    address private _owner;
    address public _newOwner;

    event OwnershipTransferred(
        address indexed previousOwner,
        address indexed newOwner);

    constructor () internal {
        _owner = msg.sender;
        emit OwnershipTransferred(address(0), _owner);
    }

    function owner() public view returns (address) {
        return _owner;
    }

    modifier onlyOwner() {
        require(isOwner());
        _;
    }

    function isOwner() public view returns (bool) {
        return msg.sender == _owner;
    }

    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
        _newOwner = address(0);
    }

    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0));
        _newOwner = newOwner;
    }

    function acceptOwnership() public {
      require(msg.sender == _newOwner);
      _transferOwnership(_newOwner);
      _newOwner = address(0);
    }

    function _transferOwnership(address newOwner) internal {
        require(newOwner != address(0));
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}
```

9. 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*.