SMART CONTRACT AUDIT REPORT

For

MAGIC

Prepared By: SFI Team

Prepared for: Magic Protocol

Prepared on: 19/09/2021

Table of Content

• Disclaimer

the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and Saferico hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, Saferico hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Saferico, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed.

• Overview of the audit

The project has 1 file. It contains approx 809 lines of Solidity code. Most of the functions and state variables are well commented on using the Nat spec documentation, but that does not create any vulnerability.

• 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 automatically.

1. Unit tests passing, checking tests configuration (matching the configuration of main network);

2. Compilator warnings;

3. Race Conditions. Reentrancy. Cross-function Race Conditions. Pitfalls in Race Condition solutions;

4. Possible delays in data delivery;

5. Transaction-Ordering Dependence (front running);

6. Timestamp Dependence;

7. Integer Overflow and Underflow;

8. DoS with (unexpected) Revert;

9. DoS with Block Gas Limit;

10. Call Depth Attack. Not relevant in modern ethereum network

11. Methods execution permissions;

12. Oracles calls;

13. Economy model. It’s important to forecast scenarios when a user is provided with additional economic motivation or faced with limitations. If application logic is based on incorrect economy model, the application will not function correctly and participants will incur financial losses. This type of issue is most often found in bonus rewards systems.
14. The impact of the exchange rate on the logic;
15. Private user data leaks.

- ## Good things in smart contract

  Compiler version is static: -
  => In this file, you have put "pragma solidity 0.6.12;" which is a good way to define the compiler version.

  => Solidity source files indicate the versions of the compiler they can be compiled with. Pragma solidity ^ 0.6.12; // bad: compiles 0.6.12 and above pragma solidity 0.6.12; //good: compiles 0.6.12 only

```
          pragma solidity 0.6.12;
```

  => If you put the (^) symbol then you are able to get compiler version 0.6.12 and above. But if you don't use the (^) symbol then you are able to use only 0.6.12 version. And if there are some changes come in the compiler and you use the old version then some issues may come at deploy time.
  => Use the latest version of solidity.

- **SafeMath library: -**
  - MAGIC are using SafeMath library it is a good thing. This protects MAGIC from underflow and overflow attacks.

```
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, with an
overflow flag.
     *
     * _Available since v3.4._
     */
```

- **Good required condition in functions: -**
  - Here you are checking that newOwner address value should be a proper and valid address.

```
function transferOwnership(address newOwner) public virtual onlyOwner
{
        require(newOwner != address(0), "Ownable: new owner is the
zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
```

  - Here you set fee , set trade, and
    set max Tx precent functions

```
function setFee( uint256 _tax,uint256 marketing,uint256 investment)
public onlyOwner{
         _taxFee = _tax;
        _investmentFee = investment;
        _marketingFee = marketing;
    }
function setTradeEnabled()public onlyOwner{
        isTradeEnabled = true;
    }

    function setMaxTxPercent(uint256 maxTxPercent) external
onlyOwner() {
        require(maxTxPercent > 0,"invalid percent");
        _maxTxAmount = _tTotal.mul(maxTxPercent).div(
            10**2
        );
    }
```

  - Here you are checking that sender and recipient addresses value should be proper and valid.

```
    function transfer(address recipient, uint256 amount) public
override returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        return true;
    }
```

  - Here you are checking that account address value should be proper and valid address.

```
    function approve(address spender, uint256 amount) public override
returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;}
```

- Here you some functions used for NFT market like (reflection From Token, exclude From Reward, include In Reward, and include Wallets In Reward)

```
function reflectionFromToken(uint256 tAmount, bool
deductTransferFee) public view returns(uint256) {
        require(tAmount <= _tTotal, "Amount must be less than
supply");
        if (!deductTransferFee) {
            (uint256 rAmount,,,,,,) = _getValues(tAmount);
            return rAmount;
        } else {
            (,uint256 rTransferAmount,,,,,) = _getValues(tAmount);
            return rTransferAmount; }
    function tokenFromReflection(uint256 rAmount) public view
returns(uint256) {
        require(rAmount <= _rTotal, "Amount must be less than total
reflections");
        uint256 currentRate =  _getRate();
        return rAmount.div(currentRate); }
    function excludeFromReward(address account) public onlyOwner()
        {if(!_isExcluded[account]){
            if(_rOwned[account] > 0) {
            _tOwned[account] =
tokenFromReflection(_rOwned[account]);}
            _isExcluded[account] = true;
            _excluded.push(account);}}
    function excludeFromReward(address[] memory accounts) public
onlyOwner{
        for(uint256 i=0; i<accounts.length;i++){
            excludeFromReward(accounts[i]);}}
    function includeInReward(address account) public onlyOwner() {
        if(_isExcluded[account]){
             for (uint256 i = 0; i < _excluded.length; i++) {
                if (_excluded[i] == account) {
                    _excluded[i] = _excluded[_excluded.length - 1];
                    _tOwned[account] = 0;
                    _isExcluded[account] = false;
                    _excluded.pop();
                    break;}}}}
    function includeWalletsInReward(address[] memory accounts)
public onlyOwner{
        for(uint256 i=0; i<accounts.length;i++){
            includeInReward(accounts[i]);}}
```

• Critical vulnerabilities found in the contract

There not Critical severity vulnerabilities found

• High vulnerabilities found in the contract

There not High severity vulnerabilities found

• Medium vulnerabilities found in the contract

```
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
require(b <= a, "SafeMath: subtraction overflow");
return a - b;
```

In detail

An overflow/underflow happens when an arithmetic operation reaches the maximum or minimum size of a type. For instance if a number is stored in the uint8 type, it means that the number is stored in a 8 bits unsigned number ranging from 0 to 2^8-1. In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of bits – either larger than the maximum or lower than the minimum representable value.

```
require(b > 0, "SafeMath: division by zero");
return a / b;
}
```

In detail

An overflow/underflow happens when an arithmetic operation reaches the maximum or minimum size of a type. For instance if a number is stored in the uint8 type, it means that the number is stored in a 8 bits unsigned number ranging from 0 to 2^8-1. In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of bits – either larger than the maximum or lower than the minimum representable value.

• Low severity vulnerabilities found

There not Low severity vulnerabilities found

• Notes

```

 uint256 private constant MAX = ~uint256(0);
uint256 private _tTotal = 10000000 * 10**18;
```

In detail
Unused variables are allowed in Solidity and they do not pose a direct security issue. It is best practice though to avoid them as they can:
- cause an increase in computations (and unnecessary gas consumption)
- indicate bugs or malformed data structures and they are generally a sign of poor code quality
- cause code noise and decrease readability of the code

• Summary of the Audit

According to automatically test, the customer`s solidity smart contract is **Secured**.

The general overview is presented in the Project Information section and all issues found are located in the audit overview section.

The test found 0 critical, 0 high, 2 medium, 0 low issues, and 1 notes.