Ether Authority

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

For

MatchNet.io (Order #22052020)

Prepared By: Chandan Kumar
Prepared For: Matchnet.io
Prepared on: 22/05/2020
Revised on: 02/06/2020
audit@etherauthority.io

Table of Content

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 of the audit

EtherAuthority (Consultant) was contracted by MatchNet (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of Customer`s smart contract and its code review conducted between May, 14th 2020 – May 22nd, 2020;
The project has five smart contract files:

• MatchToken.sol	209 lines
• RollContract.sol	327 lines
• MatchTokenDiv.sol	258 Lines
• LuxeSweep.sol	186 Lines
• LuxeSweepDiv.sol	258 Lines

It contains approx 1237 lines of Solidity code. TRC20 standard implemented properly. Logical flows are good. Safe math is also implemented correctly and other control access and security measures taken good care of. But there are some findings while auditing, where some can be ignored but some of them must be corrected and tested before production. This file is perfectly fit and recommended for production purpose, only if pointed findings are cured/checked against plan/security.

The audit was performed by two senior solidity auditors at EtherAuthority. The team has extensive work experience in developing and auditing the smart contracts.

Quick Stats:

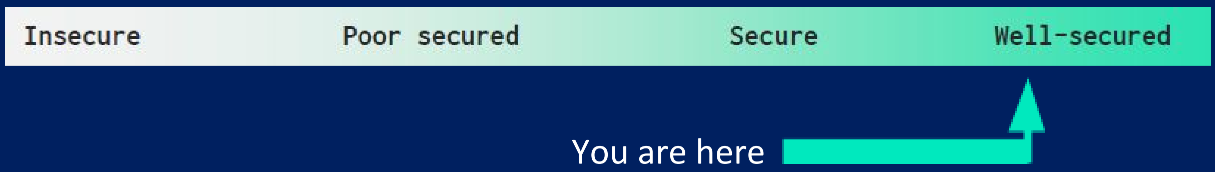
Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version is old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Passed
Code Specification	Visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	Assert() misuse	Passed
Burn	Lower limit for Burn	N/A

	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	“Out of Gas” Attack	Passed
Business Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: PASSED

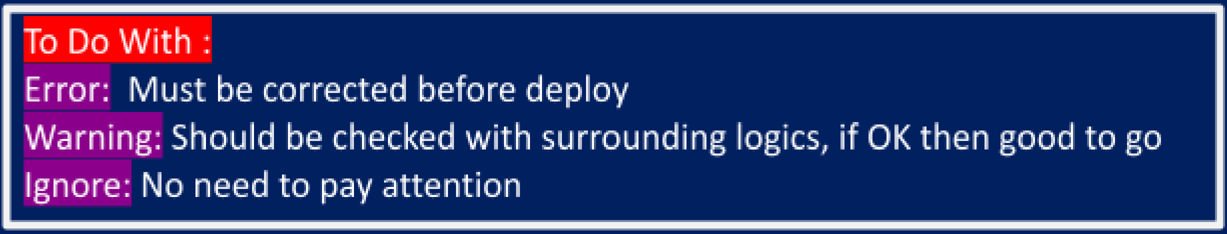
Point of Marks:

According to the assessment, Customer`s smart contract is **secured**.



Automated tools findings are as below:

This audit procedure also included the use of automated software to further scan of the code to identify potential issues:



To Do With :
Error: Must be corrected before deploy
Warning: Should be checked with surrounding logics, if OK then good to go
Ignore: No need to pay attention

https://tool.smartdec.net/scan/62f88fae08c44d4d86ed133a19ad421e
https://tool.smartdec.net/scan/bb02b74670c148769c86e2c363f64f6d

• Locked money	[Warning]
• Overpowered role	[Warning]
• Compiler version not fixed	[Ignore]
• Private modifier	[Ignore]
• Use of SafeMath	[Ignore]
• Prefer external to public visibility level	[Ignore]
• Implicit visibility level	[Warning]

https://tool.smartdec.net/scan/250df9e709c842ffb7c2352e7b3c1e27
https://tool.smartdec.net/scan/8169e38ee1bb4682baebf7c34eed4186

• Extra gas consumption	[Warning]
• Compiler version not fixed	[Ignore]
• Private modifier	[Ignore]
• Use of SafeMath	[Ignore]
• Replace multiple return values with a struct	[Ignore]
• ETH transfer inside the loop	[Warning]
• Prefer external to public visibility level	[Ignore]
• Implicit visibility level	[Warning]

https://tool.smartdec.net/scan/76d42bf023b0422084467c74415f51b2

• Multiplication after division	[Error]
• Overpowered role	[Warning]
• Compiler version not fixed	[Ignore]
• Private modifier	[Ignore]
• Use of SafeMath	[Ignore]
• Prefer external to public visibility level	[Ignore]
• Implicit visibility	[Warning]

https://mythx.io tool provided as remix.ethereum.org plugin

Above are the only few points raised by the automated tools and taken into consideration, and these are not such a problem actually for ex. loops are limited by iteration, safe math protects some attacks, and address zero is checked to move into the processing part of the function so All are OK as indicated by the above tools also.

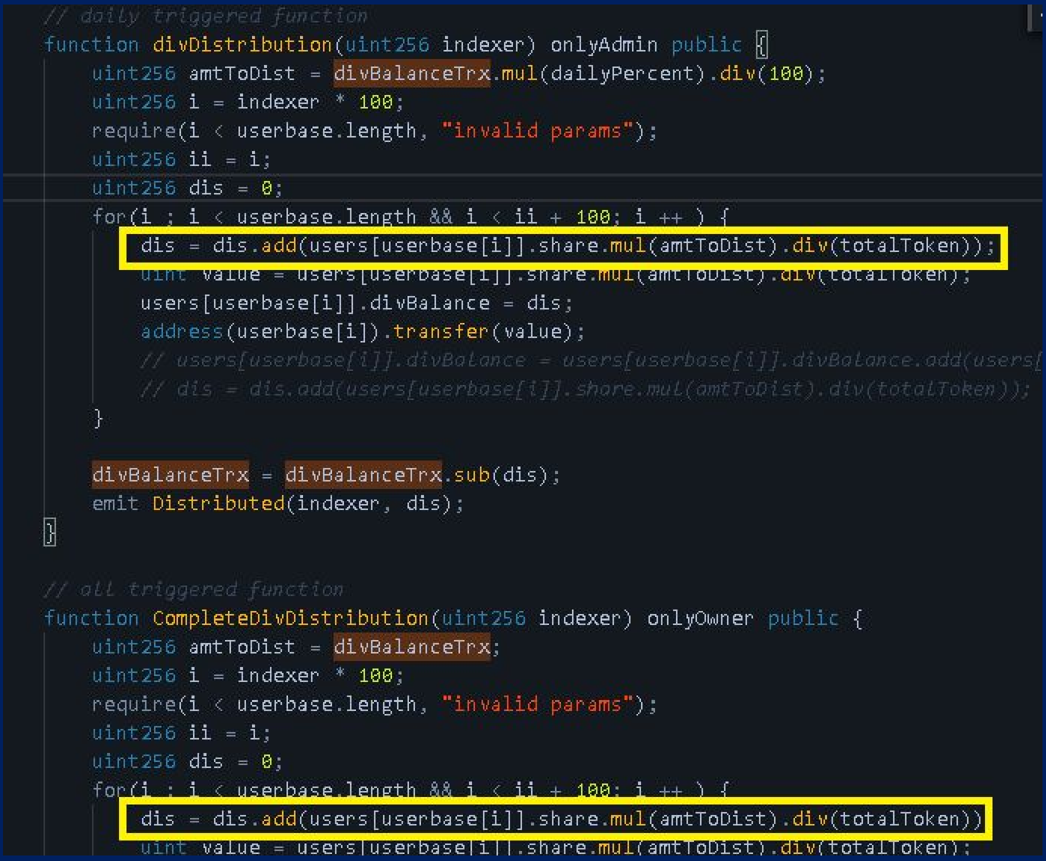
Details of Findings/Issues:

1. MatchToken.sol
    - Owner address cannot be changed later , In future it may create big trouble if this address is compromised [Rectified]
    - unlockFinds, setDivContract, updateGameContract all are payable but which is not necessary. It will create extra gas and unnecessary code size of contract. And If via these some TRX is paid to contract there is no way to withdraw those trx which is critical.[Rectified]
    - No event fired on “mine” function[Rectified]
    - “Mine” function will stop working when _totalSupply will be equal to _minedSupply if it is part of plan then OK[Rectified]
    - Storage variable “decimal = 6 ” no where used.[Rectified]
    - Funds once unlocked by the admin cannot be locked again if required for safety reasons.[Rectified]

2. LuxeSweep.sol
    - Owner address cannot be changed later , In future it may create big trouble if this address is compromised[Rectified]
    - unlockFinds, setDivContract are payable but which is not necessary. It will create extra gas and unnecessary code size of contract. And If via these some TRX is paid to contract there is no way to withdraw those trx, which is critical.[Rectified]
    - Storage variable “decimal = 6 ” no where used.[Rectified]
    - Funds once unlocked by the admin cannot be locked again if required for safety reasons.[Rectified]

3. MatchTokenDiv.sol and LuzeSweepDiv.sol
    - No upper capping on daily percent if admin set it to higher value by mistake may cause wrong value transaction to list of wallets which in turn put house in loss/extra burden on owner/admin. [Rectified]

- Calculation error may occur if (divBalanceTRX x dailyPercent) is less than 100.[Rectified]
- It will be very tedious and error prone (may double distribute to some user) due to human error for admin and if it is being done by script it is going to consume too much GAS while calling divDistribution or CompleteDivDistribution function suppose if we say the number of user is in millions, and the entire transaction cost will be too high for the owner, better to use (code) passive withdraw instead of it. It is also potential for double spend and reentrancy which may also put the house on loss. [Rectified]
- While calculating percentage fractional part is not handled properly which may lead to return 0 amount instead of valid amount on many occasions [Rectified]



```
// daily triggered function
function divDistribution(uint256 indexer) onlyAdmin public {
    uint256 amtToDist = divBalanceTrx.mul(dailyPercent).div(100);
    uint256 i = indexer * 100;
    require(i < userbase.length, "invalid params");
    uint256 ii = i;
    uint256 dis = 0;
    for(i ; i < userbase.length && i < ii + 100; i ++ ) {
        dis = dis.add(users[userbase[i]].share.mul(amtToDist).div(totalToken));
        uint value = users[userbase[i]].share.mul(amtToDist).div(totalToken);
        users[userbase[i]].divBalance = dis;
        address(userbase[i]).transfer(value);
        // users[userbase[i]].divBalance = users[userbase[i]].divBalance.add(users[
        // dis = dis.add(users[userbase[i]].share.mul(amtToDist).div(totalToken));
    }

    divBalanceTrx = divBalanceTrx.sub(dis);
    emit Distributed(indexer, dis);
}

// all triggered function
function CompleteDivDistribution(uint256 indexer) onlyOwner public {
    uint256 amtToDist = divBalanceTrx;
    uint256 i = indexer * 100;
    require(i < userbase.length, "invalid params");
    uint256 ii = i;
    uint256 dis = 0;
    for(i ; i < userbase.length && i < ii + 100; i ++ ) {
        dis = dis.add(users[userbase[i]].share.mul(amtToDist).div(totalToken))
        uint value = users[userbase[i]].share.mul(amtToDist).div(totalToken);
```

4. RollContract.sol
   - game will stop after a certain time when mined reached up to total supply [Rectified]
   - bracket missing in line 218 for combination of && with || which is critical [Rectified]
   - “userSeed[player]” is nowhere updated any value so the else block of line 254 is useless. [Rectified]
   - some of the functions has no return specified [Rectified]
   - No withdraw function found for emergency withdrawal in case some fund is stuck in contract due to logical imbalance or fractional remaining. [Rectified]

3. Attacks tested on the contract

In order to check for the security of the contract, we tested several attacks on the code. Some of those are as below:

3.1: Over and under flows

SafeMath library is used in the contract, which prevents the possibility of overflow and underflow attacks. Contract worked well under this test.

3.2: Short address attack

Although this contract is **not vulnerable** to this attack because it is good that functions are called after checking the validity of the address from the outside client.

3.3: Visibility & Delegate call

Delegate call is not used in the contract thus it does not have this vulnerability.

3.4: Reentrancy / The DAO /hack or double spend

Use of “require” function used which is good and Checks-Effects-Interactions pattern in this smart contract mitigated this vulnerability, and also some calls rooted internally like “ _transfer” are good and safe. The dividend distribution part in both DIV contracts need to shift value subtraction before transfer, because it is an admin/owner only function so it is limited by public access but still double-spend/reentrancy possible. [Rectified]

3.5: Forcing ether to a contract

Here, the Smart Contract’s balance has never been used as guard, which mitigated this vulnerability

3.6: Denial Of Service (DoS)

There is **no** any process consuming loops (if loops then limited) in the contracts which could be used for DoS attacks. and thus this contract is safe to DoS.

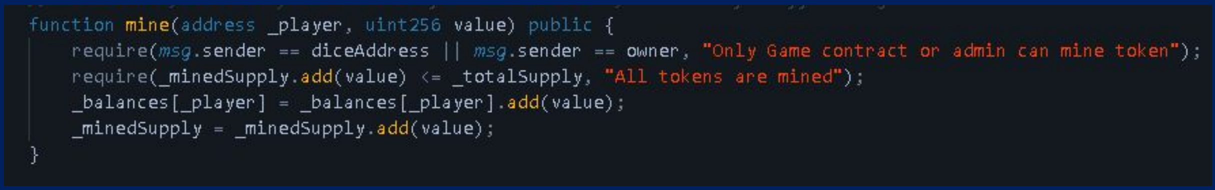
4. Good things in the smart contract

4.1 Checks-Effects-Interactions pattern

While transferring tokens, this contract does all the process first and then transfers them. The same while doing other processes too. This is very good practice which prevents malicious possibility. For example: transfer() function.

4.2 Functions input parameters passed

The functions in this contract verify the validity of the input parameters, and these validations cannot be by-passed in anyway.



```
function mine(address _player, uint256 value) public {
    require(msg.sender == diceAddress || msg.sender == owner, "Only Game contract or admin can mine token");
    require(_minedSupply.add(value) <= _totalSupply, "All tokens are mined");
    _balances[_player] = _balances[_player].add(value);
    _minedSupply = _minedSupply.add(value);
}
```

4.3 Conditions validations

The validation of input parameters is done to prevent overflow and underflow of integers. Although the SafeMath library used is also a good programming flow.

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol

4.4 More Pointers

Apart from errors and warnings (must solve before deploy), these contract are:

- Latest version (except luxeSweep which does not complied with the latest version).
- Correct implementation of TRC20 standard.
- Perfect administrative control
- Optimized for GAS.
- Good use of safe math.
- Controlled minting by admin with upper limit.
- Molecular security using “locked” features.
- **Properly commented.**

5. Critical vulnerabilities found in the contract

Critical issues that could damage heavily the integrity of the contract. Some bug that would allow attackers to steal ether is a critical issue.

- Some Mentioned in Details of Findings/Issues on page 8

Apart from that no other critical vulnerabilities were found.

6. Medium vulnerabilities found in the contract

Those vulnerabilities that could damage the contract but with some kind of limitations. Like a bug allowing people to modify a random variable.

**** No such medium vulnerabilities found in contract.**

7. Low severity vulnerabilities found

Those do not damage the contract, but better to resolve and make code clean.

- Some Mentioned in Details of Findings/Issues on page 8

**** No other low severity vulnerabilities found in contract.**

8. Gas Optimization Discussion

=> The Contract is quite good in terms of gas cost. Little more can be improved by using more optimized storage by packing multiple variables under uint256 size limit.

9. Discussions and improvements

**** Page 8, “Details of Findings” section, is the most important to focus on, to check for improvements.**

10. Summary of the Audit

After suggested modifications, this contract is safe to move on production.
It is still subject to test again after modification.

It is also encouraged to run bug bounty programs and let the community help to further polish the code to perfection.

TRC20 standard implemented properly. Logical flows are good. Safe math is also implemented correctly and other control access and security measures take good care but there are some findings while auditing , some can be ignored but some of them must be corrected and tested before production.

⇨ **So overall good to go for production.**

Ether Authority