Guild Audits

SEPTEMBER 2022

MAGIC TOKEN SMART CONTRACT AUDIT REPORT

By Guild Audits

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Description: Magic Ventures is an auto staking DeFi protocol and $MGV is the native token of Magic Ventures ecosystem. Our goal is to construct the entire ecosystem, which will be a compound combination of DeFi, NFT, Lottery, Metaverse, and other technologies.

PROJECT AUDIT SCOPE AND FINDINGS

The motive of this audit is to review the codebase of **Magic Venture** contracts for the purpose of achieving secured, corrected and quality contracts.

Number of Contracts in Scope:

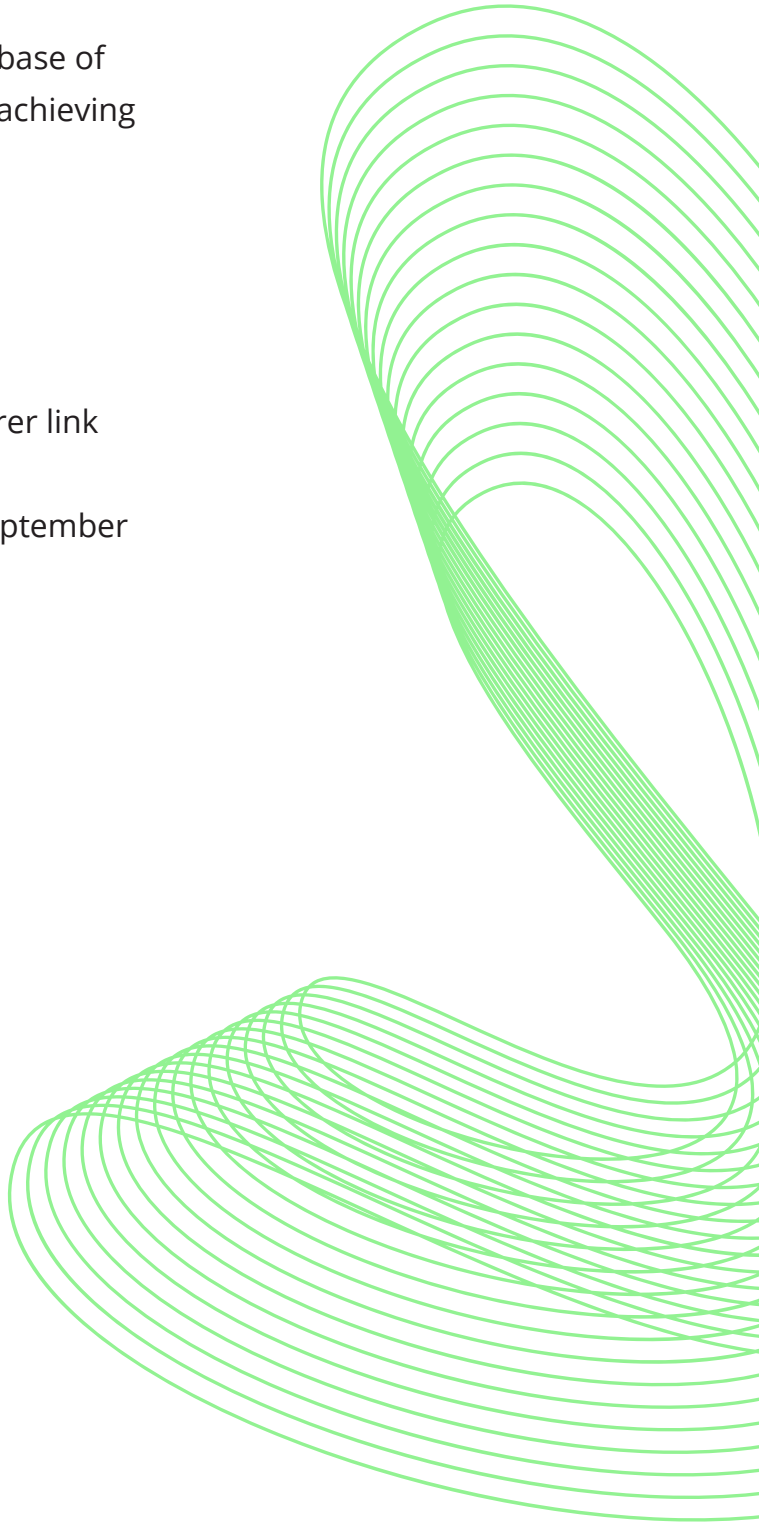
- Magic Token Contract

Link to Project codebase: Github/Block explorer link

Duration for Audit: September 01, 2022 to September 07, 2021

Audit Methodology:

Issues found:



MODE OF AUDIT AND METHODOLOGIES

The mode of audit carried out in this audit process is as follow:

- **Manual Review**: This is the first and principal step carried out to understand the business logic behind a project. At this point, it involves research, discussion and establishment of familiarity with contracts. Manual review is critical to understand the nitty-gritty of the contracts.
- **Automated Testing:** This is the running of tests with audit tools to further cement discoveries from the manual review. After a manual review of a contract, the audit tools which could be Slither, Echidna, or Mythril are run on the contract to find out issues.
- **Functional Testing:** Functional testing involves manually running unit, static, and dynamic testing. This is done to find out possible exploit scenarios that could be used to steal money from the contracts. This helps understand the functionality of the contracts and find out lapses in the reverts check in contract.

The methodologies for establishing severity issues:

- High Level Severity Issues
- Medium Level Severity Issues
- Low Level Severity Issues
- Informational Level Severity Issues

FUNCTIONAL TESTING

Some functional testing were carried out to help ascertain code security:

- Should test all the getter values
- Should be able to transfer token
- Should be able to approve
- Should be able to increase Allowance
- Should be able to decrease Allowance
- Should be able to transferFrom
- Should be able to call all onlyOwner
- Should be able to setFeeExempt and transferFrom when address isFeeExempted
- Should be able to setMaxSellTransaction and transferFrom when address

REPORT OF FINDINGS

HIGH SEVERITY ISSUES

- **Manipulating Circulating Supply**
  - Owner is allowed to set **nextRebase** to any timestamp anytime

```
1188        function setNextRebase(uint256 _nextRebase) external onlyOwner {
1189            nextRebase = _nextRebase;
1190        }
1191
```

  - Which gives owner the power to call manualRebase anytime, in order to manipulate the circulating supply.

```
1082    function manualRebase() external onlyOwner {
1083        require(!inSwap, "Try again");
1084        require(nextRebase <= block.timestamp, "Not in time");
1085
1086        uint256 circulatingSupply = getCirculatingSupply();
1087        int256 supplyDelta = int256(circulatingSupply.mul(rewardYield).div(rewardYieldDenominator));
1088
1089        coreRebase(supplyDelta);
1090        manualSync();
1091    }
1092
```

  - Which gives owner the power to call manualRebase anytime, in order to manipulate the circulating supply.

Recommendation:

  - We recommend that critical parameters like nextRebase should be changed when there is a consensus from the community or the architecture should be looked into.

Status: Resolved

- **Illogical Code**

```
806     function getLiquidityBacking(uint256 accuracy) public view returns (uint256) {
807         uint256 liquidityBalance = 0;
808         for (uint i = 0; i < _markerPairs.length; i++){
809             liquidityBalance.add(balanceOf(_markerPairs[i]).div(10 ** 9));
810         }
811         return accuracy.mul(liquidityBalance.mul(2)).div(getCirculatingSupply().div(10 ** 9));
812     }
```

Function is intended to calculate a percentage of liquidity backed up at market maker pairs corresponding to the total circulating supply. However, the function, doesn't store the liquidity balance back into the **liquidityBalance** local variable for every iteration, due to which, it will always return 0.

So, any accuracy the function will be returning 0. For the same reason

```
814     function isOverLiquified(uint256 target, uint256 accuracy) public view returns (bool) {
815         return getLiquidityBacking(accuracy) > target;
816     }
817
```

Will always return **false**, as 0 can't be greater than any **target**, not even **0**.

```
949     function swapBack() internal swapping {
950         uint256 realTotalFee = totalFee[0].add(totalFee[1]).add(totalFee[2]);
951
952         uint256 contractTokenBalance = _gonBalances[address(this)].div(_gonsPerFragment);
953
954         uint256 amountToLiquify = 0;
955         if (!isOverLiquified(targetLiquidity, targetLiquidityDenominator))
956             amountToLiquify = contractTokenBalance.mul(liquidityFee[0] + liquidityFee[1] + liquidityFee[2]).div(realTotalFee);
957         uint256 amountToMagicVault = contractTokenBalance.mul(magicVaultFee[0] + magicVaultFee[1] + magicVaultFee[2]).div(realTotalFee);
```

Which makes the condition illogical, as it will be satisfied for any given **targetLiquidity** and **targetLiquidityDenominator**.

Second Point of View:

Let's assume, the concerned function has been fixed with appropriate logic. The current **targetLiquidity** and **targetLiquidityDenominator** are 50 and 100 respectively.

Summarizing the intended logical implementation of **getLiquidityBacking** is that it will return the right percentage if given an accuracy of **50**. For instance, if the liquidity backed at market pair is **200**, and circulating supply is **500**, the percentage given will be **40**, which logically makes sense. However, the accuracy chosen by contract is **100**, which means the calculated percentage will be doubled. So, for the same example, the percentage returned now will be **80**, which will be checked against the target **50**, and even though the liquidity backed at market pairs is not actually overLiquified, the function **isOverLiquified** will return true, as **80>50**.

Recommendation:
Consider reviewing an verifying the operational and business logic, and consider fixing the implemented logic of the concerned function.

Status: Resolved

MEDIUM SEVERITY ISSUES

- **Centrallization issue**
  - The contract MagicToken.sol allows a high privileged role **owner**, to manipulate operational behaviour of the contract. However, if compromised, attacker can take advantage of the high privileged role, affecting the contract to produce unexpected results.

Recommendation:
The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts within hanced security practices, e.g., multisignature wallets.

Status: Resolved

- **Potential Issues With Tokenomics**
  - As per the description of the whitepaper and the team's response, the Titano token yields a 0.03958% interest every 30 minutes, which compounds to about 1.9176 % daily or 102, 461.12% APY. According to the code, there are 2 potential issues.1 . The rebase is not guaranteed to happen every30 minutes. The rebase will be triggered when a token transfer happens, or when the contract owner calls manualRebase(). In the case where neither of the events happen, the interest won't be added.2. The contract will stop yielding reward after a certain period of time. The compounded interest is calculated by increasing _totalSupply , which eventually adds value to users' balance. According to the code in _rebase() , the upper limit(MAX_SUPPLY) is roughly 6.805*10^10 times of

the initial_totalSupply amount. Since the APY is expected to be 102,461.12%, the amount of _totalSupply should increase by roughly 1024 times each year, and reach upper limit in 4years. Reward won't be yielded after that. Note that the contract owner can change the time interval to even shorter than 30 min, in that case the yield stop time could be earlier.f

Recommendation:
We recommend that the team be aware of the potential issues and either change the design or work around the problems

Status: Resolved

- **Owner can withdraw MagicToken**
  - Function rescueToken is used to rescue/recover tokens that are accidentally sent to magic contract. However, due to lack of input validation, function may be used by owner to extract magic tokens itself.

```
1160      function rescueToken(address tokenAddress, uint256 tokens) external onlyOwner returns (bool success) {
1161          return ERC20Detailed(tokenAddress).transfer(msg.sender, tokens);
1162      }
1163
```

Recommendation:
Consider adding checks to make sure the tokenAddress passed is not the magic token address itself.

Status: Resolved

- **Missing Input Validations**
  - Function **setRewardYield** is missing value checks for both **_rewardYield** and **_rewardYieldDenominator**, which may lead to unexpected outcomes..

```
1178      function setRewardYield(uint256 _rewardYield, uint256 _rewardYieldDenominator) external onlyOwner {
1179          rewardYield = _rewardYield;
1180          rewardYieldDenominator = _rewardYieldDenominator;
1181      }
1182
```

For instance: **_rewardYieldDenominator** may be set to 0, which may lead to divide by 0 panic. **_rewardYield** can be set to more than **_rewardYieldDenominator**, which will calculate a supply delta more than the circulating supply itself, which may not be intended.

```
1082      function manualRebase() external onlyOwner {
1083          require(!inSwap, "Try again");
1084          require(nextRebase <= block.timestamp, "Not in time");
1085
1086          uint256 circulatingSupply = getCirculatingSupply();
1087          int256 supplyDelta = int256(circulatingSupply.mul(rewardYield).div(rewardYieldDenominator));
1088
1089          coreRebase(supplyDelta);
1090          manualSync();
1091      }
```

  - Router can be set to a zero address, thereby blocking contract's operations. Also, while setting a new router, there is a need to provide allowance to the new router from the contract itself.

```
1164      function setRouterAddress(address _router) external onlyOwner {
1164          function setRouterAddress(address _router) external onlyOwner {
1165              router = IDEXRouter(_router);
1166          }
1167
```

  - A zero address can be passed as _receiver address, which will burn all the contract's AVAX liquidity

```
1155      function clearStuckBalance(address _receiver) external onlyOwner {
1156          uint256 balance = address(this).balance;
1157          payable(_receiver).transfer(balance);
1158      }
```

- whaleSellLimit and normalSellLimit can be manipulated to extract more fees from market maker pairs.

```
1200        function setWhaleSellLimit(uint256 _limit) external onlyOwner {
1201            whaleSellLimit = _limit;
1202        }
1203
1204        function setNormalSellLimit(uint256 _limit) external onlyOwner {
1205            normalSellLimit = _limit;
1206        }
```

- maxSellTransactionAmount can be manipulated to allow any amount of tokens to be transferred to market maker pairs

```
1192        function setMaxSellTransaction(uint256 _maxTxn) external onlyOwner {
1193            maxSellTransactionAmount = _maxTxn;
1194        }
1195
```

```
854
855           if (
856               automatedMarketMakerPairs[recipient] &&
857               !excludedAccount
858           ) {
859               require(amount <= maxSellTransactionAmount, "Error amount");
860           }
861
```

- **takeFee DoS:**_total can be passed any value, which means, totalFee[0] or totalFee[1] or totalFee[2] may be set to a value more than the **feeDenominator** itself.

```
1143    function setFees(uint8 _feeKind, uint256 _total, uint256 _liquidityFee, uint256 _riskFreeValue, uint256 _treasuryFee, uint256 _feeFee, uint256 _
1144        require (_liquidityFee + _riskFreeValue + _treasuryFee + _feeFee + _operationFee + _burnFee == 100, "subFee is not allowed");
1145
1146        totalFee[_feeKind] = _total * 100;
1147        liquidityFee[_feeKind] = _total * _liquidityFee;
1148        treasuryFee[_feeKind] = _total * _treasuryFee;
1149        magicVaultFee[_feeKind] = _total * _riskFreeValue;
1150        ecosystemFee[_feeKind] = _total * _feeFee;
1151        magicBankFee[_feeKind] = _total * _operationFee;
1152        burnFee[_feeKind] = _total * _burnFee;
1153    }
```

As a consequence, the takeFee function will be calculating a feeAmount more than the gonAmount itself, which will revert with underflow panic at #L1009.

```
1004
1005        uint256 feeAmount = gonAmount.mul(_realFee).div(feeDenominator);
1006
1007        _gonBalances[address(this)] = _gonBalances[address(this)].add(feeAmount);
1008        emit Transfer(sender, address(this), feeAmount.div(_gonsPerFragment));
1009        return gonAmount.sub(feeAmount);
1010    }
1011
```

- **Manipulating Business Logic:** The business logic of the token is that the _gonBalances will be storing the reflections of token for regular accounts, and actual token balances for automated market maker pairs. However, due to lack of validation any account can be set as markerPair, thus affecting the business logic, operational logic, for instance, token transfer operations, and liquidity backing calculations.

```
1093    function setAutomatedMarketMakerPair(address _pair, bool _value) public onlyOwner {
1094        require(automatedMarketMakerPairs[_pair] != _value, "Value already set");
1095
1096        automatedMarketMakerPairs[_pair] = _value;
1097
1098        if (_value) {
1099            _markerPairs.push(_pair);
1100        } else {
1101            require(_markerPairs.length > 1, "Required 1 pair");
1102            for (uint256 i = 0; i < _markerPairs.length; i++) {
1103                if (_markerPairs[i] == _pair) {
1104                    _markerPairs[i] = _markerPairs[_markerPairs.length - 1];
1105                    _markerPairs.pop();
1106                    break;
1107                }
1108            }
1109        }
```

Recommendation:
Consider adding required checks.

Status: Resolved

LOW SEVERITY ISSUES

- **No issues.**

INFORMATIONAL SEVERITY ISSUES

- **Missing Testcases**
  - Test cases for the code and functions have not been provided**.**

Recommendation:
It is recommended to write testcases of all the functions. Any existing tests that fail must be resolved. Tests will help in determining if the code is working in the expected way. Unit tests, functional tests, and integration tests should have been performed to achieve good test coverage across the entire codebase.

Status: Resolved

- **[#1114-1206] Missing Events for Critical Operations**
  - Whenever a function which is sensitive and is controlled by a centralized role it is recommended to always emit an event

Recommendation:
Consider emitting an event to functions, that are controlled by Admin(onlyOwner)

Status: Resolved

- **Missing balance checks**
  - Function **_basicTransfer** and **_transferFrom** transfer reflections/token amount Sender to receiver. However, there exist no checks to make sure the amount

being transferred should be less than or equal to sender's balance, as a consequence, the function will revert with a underflow/overflow panic code, instead of a revert message.

Recommendation:
Consider adding necessary checks

Status: Resolved

- **Unsafe downcasting from uint256 to int256**

```
1082      function manualRebase() external onlyOwner {
1083          require(!inSwap, "Try again");
1084          require(nextRebase <= block.timestamp, "Not in time");
1085
1086          uint256 circulatingSupply = getCirculatingSupply();
1087          int256 supplyDelta = int256(circulatingSupply.mul(rewardYield).div(rewardYieldDenominator));
1088
1089          coreRebase(supplyDelta);
1090          manualSync();
1091      }
1092
```

Instances of unsafe downcastings from uint256 to int256 has been reported, which may lead to unexpected results

Recommendation:
Consider using Resolvedzeppelin's SafeCast library to avoid downcasting risks

Status: Resolved

- **Unlimited Allowance:**

```
725
726         _allowedFragments[address(this)][address(router)] = ~uint256(0);
727         _allowedFragments[address(this)][pair] = ~uint256(0);
728         _allowedFragments[address(this)][address(this)] = ~uint256(0);
729
730         setAutomatedMarketMakerPair(pair, true);
731
```

The contract provides an unlimited token allowance to the router, meaning the router can extract any amount of tokens from the contract, forever, which is not a recommended practice and pattern.

Recommendation:
Consider providing a calculate limited supply, to avoid risks that may generate due to unlimited allowance.

Status: Resolved

- **Losing precision due to divide before multiply**

```
1129     function setSwapBackSettings(bool _enabled, uint256 _num, uint256 _denom) external onlyOwner {
1130         swapEnabled = _enabled;
1131         gonSwapThreshold = TOTAL_GONS.div(_denom).mul(_num);
1132     }
1133
```

Function calculates **gonSwapThreshold** using a divide before multiplying pattern, which is not a best practice and a recommended way.

Recommendation:
Consider calculating **gonSwapThreshold** by multiplying with _num first and then dividing it by **_denum**

Status: Resolved

- **Missing time margin for swapping tokens and adding liquidity**

```
924     function _addLiquidity(uint256 tokenAmount, uint256 avaxAmount) private {
925         router.addLiquidityAVAX{value: avaxAmount}(
926             address(this),
927             tokenAmount,
928             0,
929             0,
930             liquidityReceiver,
931             block.timestamp
932         );
933     }
934
935     function _swapTokensForAVAX(uint256 tokenAmount, address receiver) private {
936         address[] memory path = new address[](2);
937         path[0] = address(this);
938         path[1] = router.WAVAX();
939
940         router.swapExactTokensForAVAXSupportingFeeOnTransferTokens(
941             tokenAmount,
942             0,
943             path,
944             receiver,
945             block.timestamp
946         );
947     }
```

The contract uses a strict timestamp as a **deadline** for token swaps and adding liquidity. However, it may happen that the network is congested and the transaction may revert due to the lack of a time margin, thus it is a recommended to use a considerable time margin, for instance **block.timestamp+300**.

Note: It should be noted, that it is not recommended to give a high margin of time, as the miner may hold the transaction and add into a block at a more favourable time.

Also, it is advised to have a minimum token amount as a slippage factor instead of **0**, as it reduces the chances of DeFi Sandwich Attacks.

Recommendation:
Consider using a time margin for swaps and adding liquidity.

Status: Resolved

- **Third Party Dependencies**
  - The logic of the contract requires it to interact with third-party protocols. The scope of the audit treats 3rd party entities as black boxes and assumes their functional correctness. However, in the real world, 3rd parties can be compromised and this may lead to lost or stolen assets. In addition, upgrades of 3rd parties can possibly create severe impacts, such as increasing fees of 3rd parties, migrating to new LP pools, etc.

Recommendation:
We understand that the business logic of the MagicToken contract requires interaction with third-party protocols. We encourage the team to constantly monitor the statuses of 3rd parties to mitigate the side effects when unexpected activities are observed.

Status: Resolved

- **Missing Error Message**

```
702       modifier validRecipient(address to) {
703           require(to != address(0x0));
704           _;
705       }
706
```

Missing error message

```
1082      function manualRebase() external onlyOwner {
1083          require(!inSwap, "Try again");
1084          require(nextRebase <= block.timestamp, "Not in time");
1085
```

Error message without enough information

The require checks can be used to check for conditions and throw an exception if the condition is not met. It is better to provide an easy to understand string message containing details about the error that will be passed back to the caller

Recommendation:
We advise adding error messages to the linked require statements. Error messages, helps in debugging.

Status: Resolved

- **Multiple pragma directives have been used.**
  - There are multiple pragma directives have been used.

Recommendation:
Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (for e.g. by not using ^ in pragma solidity 0.8.4) ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs. Ref: Security Pitfall 2

Status: Resolved

- **Redundant Code**

```
1214
1215        fallback() payable external {}
```

The function **fallback() payable external {}** is use to receive ether to the contract which has been deprecated in favour of **receive() payable external {}**. Hence, It is safe to remove fallback() .

The **JustBussinessList** and **userInitialAmount** mappings are not used any where or the functions which set them.

```
763     function initialBalanceOf(address who) public view returns (uint256) {
764         return userInitialAmount[who];
765     }
766
```

```
763     function initialBalanceOf(address who) public view returns (uint256) {
764         return userInitialAmount[who];
765     }
766
```

Also in function **coreRebase**, the condition below will never be satisfied, as MAX_SUPPLY is itself the maximum value of uint256, and **_totalSupply**, also being a uint256 type, can't exceed more than that. Hence, making the condition unnecessary.

```
1069
1070         if (_totalSupply > MAX_SUPPLY) {
1071             _totalSupply = MAX_SUPPLY;
1072         }
1073
1074         _gonsPerFragment = TOTAL_GONS.div(_totalSupply);
1075
```

Recommendation:
Remove the Redundant code from the contract.

Status: Resolved

- **No need of SafeMath**
  - If the contract is intended to be deployed with solidity version 0.8.0 and above, there is no need for safeMath, as safeMath functionality has been integrated in solidity compiler version 0.8.0 and above itself.
  - However, if the compiler version for deployment is going to be less than 0.8, then there is a need to use appropriate function from SafeMath library as functions, add, sub, mul, div and mod, are plain functions and don't integrate any safe arithmetic checks for overflow and underflow scenarios, whereas alternate functions that takes an error string as a parameter, do.
  - The reason they are not giving overflow/underflow scenarios, is because, the current compiler version used is 0.8, and the unsafe

arithmetic operations are being protected by compiler itself and not SafeMath functions.

Recommendation:
Remove the RedundConsider removing SafeMath dependency if 0.8 is the intended version for deployment, if the intended compiler is supposed to be less than 0.8, then there is a need to adopt correct SafeMath functions.ant code from the contract.

Status: Resolved

- **Naming conventions**
  - Functions or variable should be given name that tell or interpret what it should do

```
646
647     mapping (address => bool) _isFeeExempt;
648     address[] public _markerPairs;
649     mapping (address => bool) public automatedMarketMakerPairs;
650     mapping (address => bool) public justBusinessList;
```

Recommendation:
Consider using a more easy to understand name like marketPair or makerPair

Status: Resolved

- **ERC20 approve() race**
  - The standard ERC20 implementation contains a widely-known racing condition in its approve function, wherein a spender is able to witness the token owner broadcast a transaction altering their approval, and quickly sign and broadcast a transaction using transferFrom to move the current approved amount from the

owner's balance to the spender. If the spender's transaction is validated before the owner's, the spender is able to spend their entire approval amount twice.

Reference:
https://eips.ethereum.org/EIPS/eip-20

CLOSING SUMMARY
There were discoveries of some high, medium, low and informational issues after the audit. The audit team thereafter suggested some remediation to help remedy the issues found in the contract.

APPENDIX
- *Audit: The review and testing of contracts to find bugs.*
- *Issues: These are possible bugs that could lead exploits or help redefine the contracts better.*
- *Slither: A tool used to automatically find bugs in a contract.*
- *Severity: This explains the status of a bug.*

DISCLAIMER
While the audit report is aimed at achieving a quality codebase with assured security and correctness, it should not be interpreted as a guide or or recommendation for people to invest in **Magic Venture** contracts.
With smart contract audit being a multifaceted process, we admonish the **Magic Venture** team to carry out further audit from other audit firms or provide a bug bounty program to ensure that more critical audit is done to the contract.

GUILD AUDITS
Guild Audits is geared towards providing blockchain and smart contract security in the fuming web3 world. The firm is passionate about remedying the constant hacks and exploits that deters the web3 motive.