MMPRO Token

Smart Contract Audit Report

SYSFIXED

21 Oct 2021

TABLE OF CONTENTS

AUDITED DETAILS

Audited Project

Project name	Token ticker	Blockchain
MMPRO Token	MMPRO	Binance Smart Chain

Addresses

Contract address	0x6067490d05f3cf2fdffc0e353b1f5fd6e5ccdf70
Contract deployer address	0x1Cc23586AeA610Df8e9a850B7393a1251A30EA8A

Project Website

https://marketmaking.pro/

Codebase

https://bscscan.com/address/0x6067490d05f3cf2fdffc0e353b1f5fd6e5ccdf70#code

SUMMARY

MMPRO Token is a part of our whole ecosystem—market making. Pro is an agency that provides market-making and liquidity services to exchanges and projects. Besides that, MMPRO is building a cross-chain multi-farming ecosystem.

Contract Summary

Documentation Quality

MMPRO Token provides a very poor documentation with standard of solidity base code.

- The technical description is provided unclear and disorganized.

Code Quality

The Overall quality of the basecode is poor.

- Solidity basecode and rules are unclear and disorganized by MMPRO Token.

Test Coverage

Test coverage of the project is 100% (Through Codebase)

Audit Findings Summary

- SWC-100 SWC-108 | Explicitly define visibility for all state variables on lines 68, 211, 218, 238, 101 and 155.
- SWC-103 | Pragma statements can be allowed to float when a contract is intended on lines 5.
- SWC-110 SWC-123 | It is recommended to use of revert(), assert(), and require() in Solidity, and the new REVERT opcode in the EVM on lines 103 and 98.
- SWC-111 | It is recommended to use alternatives to the deprecated constructions on lines 30, 36, 43, 48, 89, 130, 140 and 201.

CONCLUSION

We have audited the MMPRO Token project released on October 2021 to find issues and identify potential security vulnerabilities in the MMPRO Token project. This process is used to find technical issues and security loopholes that may be found in smart contracts.

The security audit report yielded unsatisfactory results, discovering medium-risk and low-risk issues.

Writing a contract that does not follow the Solidity style guide can pose a significant risk. The serious and low problems we found in the smart contract are An assertion violation triggered, a low-risk issue, a floating pragma is set, and state variable visibility is not set. The smart contract delegates execution to a user-supplied address. It is possible to trigger an assertion violation. Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (via passed arguments) and callees (for instance, via return values). A floating pragma is set, and the current pragma Solidity directive is ""^0.4.24"". Specifying a fixed compiler version is recommended to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

We were recommended to keep being aware of investing in this risky smart contract.

AUDIT RESULT

Article	Category	Description	Result
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	ISSUE FOUND
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	PASS
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the Solidity compiler.	PASS
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	ISSUE FOUND
Unchecked Call Return Value	SWC-104	The return value of a message call should be checked.	PASS
Unprotected Ether Withdrawal	SWC-105	Due to missing or insufficient access controls, malicious parties can withdraw from the contract.	PASS
SELFDESTRUCT Instruction	SWC-106	The contract should not be self-destructible while it has funds belonging to users.	PASS
Reentrancy	SWC-107	Check effect interaction pattern should be followed if the code performs recursive call.	PASS
Uninitialized Storage Pointer	SWC-109	Uninitialized local storage variables can point to unexpected storage locations in the contract.	PASS
Assert Violation	SWC-110 SWC-123	Properly functioning code should never reach a failing assert statement.	ISSUE FOUND
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	ISSUE FOUND
Delegate call to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	PASS

DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless required.	PASS
Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	PASS
Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	PASS
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	PASS
Signature Unique ID	SWC-117 SWC-121 SWC-122	Signed messages should always have a unique id. A transaction hash should not be used as a unique id.	PASS
Incorrect Constructor Name	SWC-118	Constructors are special functions that are called only once during the contract creation.	PASS
Shadowing State Variable	SWC-119	State variables should not be shadowed.	PASS
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes or be predictable.	PASS
Write to Arbitrary Storage Location	SWC-124	The contract is responsible for ensuring that only authorized user or contract accounts may write to sensitive storage locations.	PASS
Incorrect Inheritance Order	SWC-125	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order. The rule of thumb is to inherit contracts from more /general/ to more /specific/.	PASS
Insufficient Gas Griefing	SWC-126	Insufficient gas griefing attacks can be performed on contracts which accept data and use it in a sub-call on another contract.	PASS
Arbitrary Jump Function	SWC-127	As Solidity doesnt support pointer arithmetics, it is impossible to change such variable to an arbitrary value.	PASS

Typographical Error	SWC-129	A typographical error can occur for example when the intent of a defined operation is to sum a number to a variable.	PASS
Override control character	SWC-130	Malicious actors can use the Right-To-Left-Override unicode character to force RTL text rendering and confuse users as to the real intent of a contract.	PASS
Unused variables	SWC-131 SWC-135	Unused variables are allowed in Solidity and they do not pose a direct security issue.	PASS
Unexpected Ether balance	SWC-132	Contracts can behave erroneously when they strictly assume a specific Ether balance.	PASS
Hash Collisions Variable	SWC-133	Using abi.encodePacked() with multiple variable length arguments can, in certain situations, lead to a hash collision.	PASS
Hardcoded gas amount	SWC-134	The transfer() and send() functions forward a fixed amount of 2300 gas.	PASS
Unencrypted Private Data	SWC-136	It is a common misconception that private type variables cannot be read.	PASS

SMART CONTRACT ANALYSIS

Started	Wednesday Oct 20 2021 21:53:33 GMT+0000 (Coordinated Universal Time)
Finished	Thursday Oct 21 2021 16:16:05 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Main Source File	MMPRO.sol

Detected Issues

ID	Title	Severity	Status
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	medium	acknowledged
SWC-100	FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)	low	acknowledged
SWC-100	FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)	low	acknowledged
SWC-100	FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)	low	acknowledged
SWC-100	FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-108	STATE VARIABLE VISIBILITY IS NOT SET.	low	acknowledged
SWC-108	STATE VARIABLE VISIBILITY IS NOT SET.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged

SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 103

medium SEVERITY

It is possible to trigger an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- MMPRO.sol

Locations

```
102
103   bool public locked = true;
104
105   function setLocked(bool _locked) public onlyOwner {
106   locked = _locked;
107
```

SWC-100 | FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)

LINE 68

low SEVERITY

The function definition of "Ownable" lacks a visibility specifier. Note that the compiler assumes "public" visibility by default. Function visibility should always be specified explicitly to assure correctness of the code and improve readability.

Source File

- MMPRO.sol

Locations

```
67   */
68   function Ownable() {
69   owner = msg.sender;
70   }
71
72
```

SWC-100 | FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)

LINE 211

low SEVERITY

The function definition of "increaseApproval" lacks a visibility specifier. Note that the compiler assumes "public" visibility by default. Function visibility should always be specified explicitly to assure correctness of the code and improve readability.

Source File

- MMPRO.sol

Locations

```
210   */
211   function increaseApproval (address _spender, uint _addedValue)
212   returns (bool success) {
213   allowed[msg.sender][_spender] = allowed[msg.sender][_spender].add(_addedValue);
214   Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
215
```

SWC-100 | FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)

LINE 218

low SEVERITY

The function definition of "decreaseApproval" lacks a visibility specifier. Note that the compiler assumes "public" visibility by default. Function visibility should always be specified explicitly to assure correctness of the code and improve readability.

Source File

- MMPRO.sol

Locations

```
217
218   function decreaseApproval (address _spender, uint _subtractedValue)
219   returns (bool success) {
220   uint oldValue = allowed[msg.sender][_spender];
221   if (_subtractedValue > oldValue) {
222
```

SWC-100 | FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)

LINE 238

low SEVERITY

The function definition of "MMPRO" lacks a visibility specifier. Note that the compiler assumes "public" visibility by default. Function visibility should always be specified explicitly to assure correctness of the code and improve readability.

Source File

- MMPRO.sol

Locations

```
237
238   function MMPRO () {
239   totalSupply = initialSupply;
240   balances[msg.sender] = initialSupply;
241
242
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 5

low SEVERITY

The current pragma Solidity directive is ""^0.4.24"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- MMPRO.sol

Locations

```
4
5   pragma solidity ^0.4.24;
6
7   /**
8   ********************************
9
```

SWC-108 | STATE VARIABLE VISIBILITY IS NOT SET.

LINE 101

low SEVERITY

It is best practice to set the visibility of state variables explicitly. The default visibility for "balances" is internal. Other possible visibility settings are public and private.

Source File

- MMPRO.sol

Locations

```
100
101   mapping(address => uint256) balances;
102
103   bool public locked = true;
104
105
```

SWC-108 | STATE VARIABLE VISIBILITY IS NOT SET.

LINE 155

low SEVERITY

It is best practice to set the visibility of state variables explicitly. The default visibility for "allowed" is internal. Other possible visibility settings are public and private.

Source File

- MMPRO.sol

Locations

```
154
155   mapping (address => mapping (address => uint256)) allowed;
156
157   /**
158   * @dev Transfer tokens from one address to another
159
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 98

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- MMPRO.sol

Locations

```
97   */
98   contract BasicToken is ERC20Basic, Ownable {
99   using SafeMath for uint256;
100
101   mapping(address => uint256) balances;
102
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 30

low SEVERITY

Using "constant" as a state mutability modifier in function "mul" is disallowed as of Solidity version 0.5.0. Use "view" instead.

Source File

- MMPRO.sol

Locations

```
29   library SafeMath {
30   function mul(uint256 a, uint256 b) internal constant returns (uint256) {
31   uint256 c = a * b;
32   assert(a == 0 || c / a == b);
33   return c;
34
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 36

low SEVERITY

Using "constant" as a state mutability modifier in function "div" is disallowed as of Solidity version 0.5.0. Use "view" instead.

Source File

- MMPRO.sol

Locations

```
35
36   function div(uint256 a, uint256 b) internal constant returns (uint256) {
37   // assert(b > 0); // Solidity automatically throws when dividing by 0
38   uint256 c = a / b;
39   // assert(a == b * c + a % b); // There is no case in which this doesn't hold
40
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 43

low SEVERITY

Using "constant" as a state mutability modifier in function "sub" is disallowed as of Solidity version 0.5.0. Use "view" instead.

Source File

- MMPRO.sol

Locations

```
42
43   function sub(uint256 a, uint256 b) internal constant returns (uint256) {
44   assert(b <= a);
45   return a - b;
46   }
47
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 48

low SEVERITY

Using "constant" as a state mutability modifier in function "add" is disallowed as of Solidity version 0.5.0. Use "view" instead.

Source File

- MMPRO.sol

Locations

```
47
48   function add(uint256 a, uint256 b) internal constant returns (uint256) {
49   uint256 c = a + b;
50   assert(c >= a);
51   return c;
52
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 89

low SEVERITY

Using "constant" as a state mutability modifier in function "balanceOf" is disallowed as of Solidity version 0.5.0. Use "view" instead.

Source File

- MMPRO.sol

Locations

```
88   uint256 public totalSupply;
89   function balanceOf(address who) public constant returns (uint256);
90   function transfer(address to, uint256 value) public returns (bool);
91   event Transfer(address indexed from, address indexed to, uint256 value);
92   }
93
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 130

low SEVERITY

Using "constant" as a state mutability modifier in function "balanceOf" is disallowed as of Solidity version 0.5.0. Use "view" instead.

Source File

- MMPRO.sol

Locations

```
129   */
130   function balanceOf(address _owner) public constant returns (uint256 balance) {
131   return balances[_owner];
132   }
133   }
134
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 140

low SEVERITY

Using "constant" as a state mutability modifier in function "allowance" is disallowed as of Solidity version 0.5.0. Use "view" instead.

Source File

- MMPRO.sol

Locations

```
139   contract ERC20 is ERC20Basic {
140   function allowance(address owner, address spender) public constant returns
(uint256);
141   function transferFrom(address from, address to, uint256 value) public returns
(bool);
142   function approve(address spender, uint256 value) public returns (bool);
143   event Approval(address indexed owner, address indexed spender, uint256 value);
144
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 201

low SEVERITY

Using "constant" as a state mutability modifier in function "allowance" is disallowed as of Solidity version 0.5.0. Use "view" instead.

Source File

- MMPRO.sol

Locations

```
200   */
201   function allowance(address _owner, address _spender) public constant returns
(uint256 remaining) {
202   return allowed[_owner][_spender];
203   }
204
205
```

DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to, or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Sysfixed's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Sysfixed to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model, or legal compliance.

This is a limited report on our findings based on our analysis, in accordance with good industry practice as of the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the below disclaimer below – please make sure to read it in full.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Sysfixed and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (Sysfixed) owe no duty of care.

ABOUT US

Sysfixed is a blockchain security certification organization established in 2021 with the objective to provide smart contract security services and verify their correctness in blockchain-based protocols. Sysfixed automatically scans for security vulnerabilities in Ethereum and other EVM-based blockchain smart contracts. Sysfixed a comprehensive range of analysis techniques—including static analysis, dynamic analysis, and symbolic execution—can accurately detect security vulnerabilities to provide an in-depth analysis report. With a vibrant ecosystem of world-class integration partners that amplify developer productivity, Sysfixed can be utilized in all phases of your project's lifecycle. Our team of security experts is dedicated to the research and improvement of our tools and techniques used to fortify your code.