



MATH Token

Smart Contract Audit Report

TABLE OF CONTENTS

[Audited Details](#)

- Audited Project
- Blockchain
- Addresses
- Project Website
- Codebase

[Summary](#)

- Contract Summary
- Audit Findings Summary
- Vulnerabilities Summary

[Conclusion](#)

[Audit Results](#)

[Smart Contract Analysis](#)

- Detected Vulnerabilities

[Disclaimer](#)

[About Us](#)

AUDITED DETAILS

Audited Project

| Project name | Token ticker | Blockchain |
|--------------|--------------|------------|
| MATH Token | MATH | Ethereum |

Addresses

| | |
|---------------------------|--------------------------------------------|
| Contract address | 0x08d967bb0134f2d07f7cfb6e246680c53927dd30 |
| Contract deployer address | 0x413B51f2DD56D3C1BE678FF074a77b55D39ab8e2 |

Project Website

<https://mathwallet.org/en-us/>

Codebase

<https://etherscan.io/address/0x08d967bb0134f2d07f7cfb6e246680c53927dd30#code>

SUMMARY

MATH is a one-stop crypto platform that combines several applications, including Math Wallet, MATH VPOS Pool, Math DApp Store, MathStaking, MathPay and MathChain. It was founded in 2018.

Contract Summary

Documentation Quality

MATH 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 MATH 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 77 and 79.
- SWC-103 | Pragma statements can be allowed to float when a contract is intended on lines 9.
- 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 53.

CONCLUSION

We have audited the MATH Token token project released in September 2019 to find issues and identify potential security vulnerabilities in the MATH Token token project. This process is used to find technical issues and security loopholes that may be found in smart contracts.

The security audit report gave unsatisfactory results with the discovery of a medium-risk issue and other low-risk issues.

Writing a contract that does not follow the Solidity style guide can pose a significant risk. The medium risk issue we found is an assertion violation was triggered. 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). Whereas the low risk Issues we found are a state variable visibility is not set and a floating pragma is set. 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.

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. | PASS |
| 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 <code>abi.encodePacked()</code> with multiple variable length arguments can, in certain situations, lead to a hash collision. | PASS |
| Hardcoded gas amount | SWC-134 | The <code>transfer()</code> and <code>send()</code> 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 | Friday Sep 27 2019 18:06:12 GMT+0000 (Coordinated Universal Time) |
| Finished | Saturday Sep 28 2019 13:18:32 GMT+0000 (Coordinated Universal Time) |
| Mode | Standard |
| Main Source File | MATHToken.sol |

Detected Issues

| ID | Title | Severity | Status |
|---------|---------------------------------------|----------|--------------|
| SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED. | medium | 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.

LINE 53

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

- MATHToken.sol

Locations

```
52  c = a + b;  
53  assert(c >= a);  
54  return c;  
55  }  
56  }  
57
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 9

low SEVERITY

The current pragma Solidity directive is `^0.4.23`. 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

- MATHToken.sol

Locations

```
8
9  pragma solidity ^0.4.23;
10
11 // based on https://github.com/OpenZeppelin/openzeppelin-solidity/tree/v1.10.0
12 /**
13
```

SWC-108 | STATE VARIABLE VISIBILITY IS NOT SET.

LINE 77

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

- MATHToken.sol

Locations

```
76
77 mapping(address => uint256) balances;
78
79 uint256 totalSupply_;
80
81
```

SWC-108 | STATE VARIABLE VISIBILITY IS NOT SET.

LINE 79

low SEVERITY

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

Source File

- MATHToken.sol

Locations

```
78
79  uint256 totalSupply_;
80
81  /**
82   * @dev total number of tokens in existence
83
```

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.