MATHMASTERS

SECURITY AUDIT REPORT::2024-02-08

SAFIRAETHER.COM
SAFEGUARDING ETHEREUM

Prepared by 808Nestor
NESTOR@SAFIRAETHER.COM

TABLE OF CONTENTS

INTRODUCTION

About The Protocol

Protocol Objective

Math Master is a specialized protocol created by the math MASTERS, focusing on efficient and algebraically sound web3 development. Their library, MathMasters, offers optimized functions like WAD for basic quantities, mulWad for rounded-down multiplication, mulWadUp for rounded-up multiplication, and sqrt for square roots. The protocol prioritizes EVM/assembly level integrity, specifying low severity issues, and encourages testing for compatibility with Solidity projects using version ^0.8.3.

Solc Version

0.8.3 < 0.9.0

Chains

Ethereum

Tokens

None

Audit Scope

Commit Hash

None

Files in Scope

MathMasters.sol

Audit Roles

None

Known Issues

None

DISCLAIMER

All actions by The Audit Team in this project adhere to the agreed statement of work and project plan. Security assessments are time-limited and rely on client-provided information. The findings in this report may not cover all security issues in the target system or codebase.

Automated testing supplements manual security reviews but has limitations. Tools may not cover all edge cases within the allocated time. This audit doesn't replace functional tests or guarantee identifying all security issues, emphasizing the need for multiple audits and a bug bounty program.

This report isn't investment advice and is subject to the terms of the Services Agreement. Distribution or reliance by any party other than the designated client is prohibited without prior written consent.

It neither endorses nor disapproves of any project, providing no insight into economic value or legal compliance. Users access services at their risk, acknowledging uncertainties of cryptographic tokens and blockchain technology.

The assessment may not uncover all vulnerabilities, and the absence of identified issues doesn't ensure a secure system. The Audit Team focuses on source code assessments, acknowledging software development limitations and potential impacts of third-party infrastructure.

The Audit Team, dedicated to discovering vulnerabilities within a timeframe, doesn't assume responsibility for findings outlined in this document. The audit solely addresses solidity implementation, not endorsing the underlying business. Recognizing time constraints, it exclusively focuses on the security aspects of the assessed code.

METHODOLOGY

Our audit methodology focused on ensuring the security, reliability, and adherence to best practices in the context of the Ethereum blockchain.

BUSINESS LOGIC ASSESSMENT

Our analysis commenced with a thorough understanding of the smart contract's business logic. The goal was to identify the core functionalities and interactions with external components, laying the foundation for subsequent evaluations.

MANUAL CODE REVIEW

A meticulous review of the Solidity source code was conducted, adhering to industry best practices and coding standards. The purpose was to identify potential vulnerabilities, ensuring the code is robust and maintainable.

AUTOMATED ANALYSIS

Advanced automated analysis tools, including Aderyn, and Slither, were employed to identify common vulnerabilities related to security, gas efficiency, and code style. The results contributed to a comprehensive understanding of potential risks. Additionally, testing of invariants was conducted where appropriate via fuzzing and formal verification using tools such as Halmos and Certora.

SECURITY PATTERNS APPLICATION

Security patterns and anti-patterns were applied to address common vulnerabilities, including reentrancy, overflow/underflow, and timestamp dependency. Access controls and permissions were implemented judiciously to enhance overall security.

EXTERNAL CALLS EVALUATION

External calls to other contracts or external systems were scrutinized to mitigate the risk of reentrancy attacks. The assessment ensured that all calls were secure and aligned with the integrity of the contract.

INPUT VALIDATION

Rigorous validation of user inputs was conducted to ensure the smart contract gracefully handles unexpected inputs, guarding against vulnerabilities such as integer overflow and underflow.

Risk Classifications

Risk is classified based on two factors: likelihood and impact.

Likelihood

Likelihood refers to the probability of a specific event or vulnerability being exploited. Here's an elaboration on the three levels of likelihood:

High Likelihood:

In situations of high likelihood, the conditions for exploitation are readily accessible or the attack vector is easily achievable. The vulnerability is considered highly exploitable, and the likelihood of occurrence is relatively high.

Example Scenario: A scenario where a hacker can directly call a function to cause a significant impact on the smart contract's behavior. This could involve straightforward and easily executable steps that do not require elaborate conditions.

Medium Likelihood:

In cases of medium likelihood, specific conditions or a more constrained set of circumstances are required for the vulnerability to be exploited. While not as easily achievable as high likelihood scenarios, the conditions for exploitation are still reasonably attainable, making the event moderately likely to occur.

Example Scenario: An example could be a vulnerability that depends on the use of a particular type of token within the platform. While not universally applicable, the conditions for exploitation are plausible and could occur under specific circumstances.

Low Likelihood:

Vulnerabilities with low likelihood are associated with rare situations that are unlikely to happen in typical scenarios. Although technically feasible, the conditions required for exploitation are infrequent or involve a combination of events that are unlikely to align.

Example Scenario: Consider a vulnerability that relies on a unique sequence of events (A, B, C) taking place at a specific time. While technically possible, the occurrence of such a sequence is rare and unlikely, making the exploitation of the vulnerability less probable.

NOTE:

Computationally Unfeasible: Events that are 'computationally unfeasible' are practically impossible due to their extreme rarity or the astronomical computational effort required for exploitation. These are not considered viable attack paths in practice.

Assessing likelihood involves a degree of subjectivity and requires a comprehensive understanding of the smart contract's architecture, the blockchain environment, and potential threat vectors. Regular reviews and updates to likelihood assessments should be conducted to adapt to changes in the threat landscape and the evolving nature of decentralized applications.

IMPACT

Impact is a crucial aspect of vulnerability assessment, representing the potential harm or consequences resulting from a vulnerability. Here are the three levels of impact:

HIGH IMPACT:

High impact vulnerabilities pose a significant threat to the protocol, where funds are directly or nearly directly at risk. The consequences involve a severe disruption of protocol functionality or availability, with potential financial losses for users.

Examples:

- Direct exposure of funds to unauthorized access.
- Severe disruption in the protocol's core functionality, leading to financial losses.

MEDIUM IMPACT:

Medium impact vulnerabilities introduce some level of risk to funds, albeit indirectly. While not as severe as high impact scenarios, these vulnerabilities can still result in disruptions to the protocol's functionality or availability, impacting the user experience and potentially leading to financial consequences.

Examples:

- Indirect risk to funds due to a vulnerability that affects the integrity of transactions or user accounts.
- Moderate disruption in protocol functionality, affecting user interactions and potentially leading to financial consequences.

Low Impact:

Low impact vulnerabilities do not directly put funds at risk. However, there might be issues related to the correctness of functions, inappropriate handling of states, or other non-financial implications. The primary concern is related to the correctness and reliability of the protocol rather than immediate financial losses.

Examples:

- Correctness issues in specific functions that do not pose a direct threat to funds but may impact the overall reliability of the protocol.
- State handling concerns that do not immediately impact financial transactions but may affect the protocol's overall performance.

Assessing impact is crucial for prioritizing the resolution of vulnerabilities and allocating resources effectively. It helps in understanding the potential consequences of each vulnerability and guides the development team in addressing the most critical issues first, ensuring the overall security and stability of the protocol. Regular impact assessments should be conducted to adapt to changes in the protocol's features and user interactions.

Severity Ratings

Severity ratings in the context of smart contract security assessments are typically derived from the combination of likelihood and impact assessments. Here are the severity ratings based on the provided likelihood and impact descriptions:

Impact	Low	Medium	High
High	MEDIUM	HIGH	CRITICAL
Medium	LOW	MEDIUM	HIGH
Low	LOW	LOW	MEDIUM

Likelihood

Critical Severity:

- High Likelihood + High Impact

Vulnerabilities with a critical severity rating represent a high likelihood of exploitation coupled with significant consequences. In these scenarios, funds are directly at risk, and there's a severe disruption of protocol functionality or availability, posing a substantial threat to users and the overall integrity of the protocol.

HIGH SEVERITY:

- Medium Likelihood + High Impact
- High Likelihood + Medium Impact

High severity vulnerabilities indicate a substantial risk to the protocol. While the likelihood may be high or medium, the impact is never low, with the potential for direct financial losses and severe disruption of protocol functionality. These vulnerabilities demand immediate attention and remediation efforts.

MEDIUM SEVERITY:

- Low Likelihood + High Impact
- Medium Likelihood + Medium Impact
- High Likelihood + Low Impact

Medium severity vulnerabilities pose a moderate risk to the protocol. While the likelihood of exploitation and the impact may vary, the potential for indirect financial risks and disruptions to the protocol's functionality remains constant. Timely remediation is recommended to maintain the overall security and stability of the system.

LOW SEVERITY:

- Low Likelihood + Low Impact
- Low Likelihood + Medium Impact
- Medium Likelihood + Low Impact

Low severity vulnerabilities represent a lower risk to the protocol. The likelihood of exploitation is low or medium, and the impact may be low or medium, primarily related to correctness issues or non-financial implications. While these vulnerabilities are not immediate threats, they should still be addressed in a timely manner to enhance the robustness of the protocol.

INFORMATIONAL SEVERITY:

- Likelihood: Not Applicable
- Impact: Not Applicable

Informational severity is used for findings that provide valuable information but do not pose an immediate risk to the protocol's security or functionality. These findings may include suggestions for improvement or best practices that could enhance the overall security posture.

FINDINGS

EXECUTIVE SUMMARY

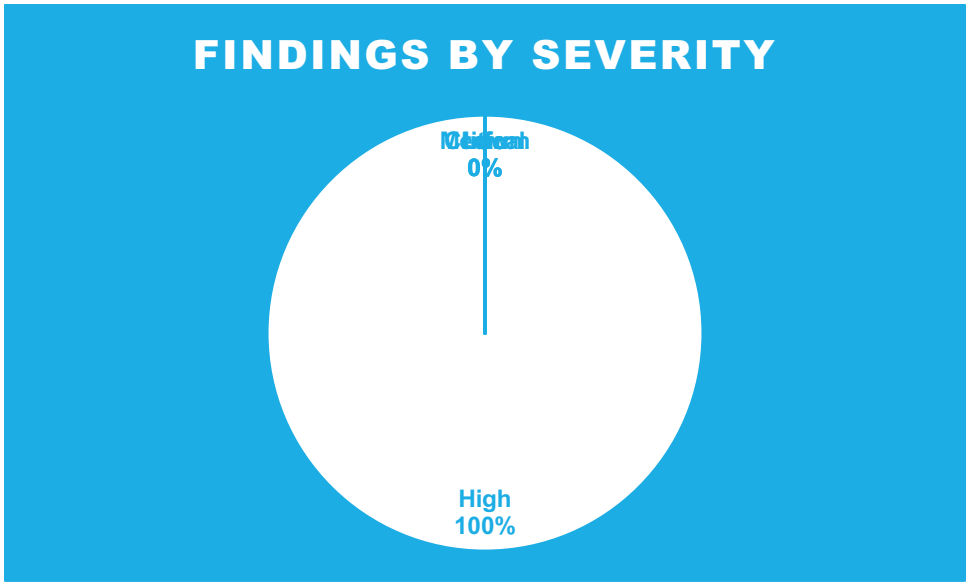
OVERVIEW

The smart contract audit report revealed two high-severity findings that require immediate attention. In the first case (H-01), the MathMasters::mulWadUp() function's overflow check is flawed. While attempting to prevent overflow in fixed-point arithmetic, the use of the OR operator itself may lead to an overflow. Specifically, the conditions check if certain values are non-zero and greater than predefined limits, but the OR operator introduces a vulnerability. This finding poses a high risk to the smart contract's integrity. The second issue (H-02) pertains to the MathMasters::sqrt() function, which inaccurately uses decimal notation for constants instead of the recommended hexadecimal representation in Ethereum's Solidity programming language. Hexadecimal notation is crucial for maintaining precision, especially in low-level constructs like assembly. The mismatch in representation between the MathMasters implementation and the expected hexadecimal values by the Ethereum Virtual Machine results in a significant loss of precision, emphasizing the need for prompt corrective action.

FINDINGS BY SEVERITY TALLY TABLE

Severity	Tally
Critical	0
High	2
Medium	0
Low	0
Info	0
TOTAL	2

FINDINGS BY SEVERITY TALLY CHART



Critical Severity Findings

N/A

High Severity Findings

H-01: Use of OR in MathMasters::mulWadUp() overflow check results in overflow.

File: MathMasters

Element: mulWadUp()

Likelihood: Medium

Financial Impact: High

Severity: High

Details

The function checks for potential overflow in the context of fixed-point arithmetic by using the OR operator. However, the OR operator itself can cause an overflow. In this case, the first condition checks if y is non-zero and if x is greater than the maximum value divided by y. The second condition checks if x is non-zero.

Impact

In situations where x is 2 and y is 5.789e76, an overflow will occur.

Proof Of Concept

```
function test_fuzz_mulWadUp(uint256 fuzzX, uint256 fuzzY) public {
        // ARRANGE: specify input conditions
        uint256 x = fuzzX;
        uint256 y = fuzzY;

        // ACT: call target contracts
        uint256 z = MathMasters.mulWadUp(x, y);

        uint256 solution = (x * y) / (1e18);
        if (solution * 1e18 < x * y) {
```

```
            solution += 1;
        }
        // ASSERT: check output states
        assertEq(z, solution);
    }
```

Tools Used

Forge

Recommendations

Use a different method to check for potential overflow or do not check for x being non-zero. The equivalent function in solady, for example, does not check if x is non-zero in its overflow check.

```
- if mul(y, gt(x, or(div(not(0), y), x))) {
+ if mul(y, gt(x, div(not(0), y))) {
```

H-02: Use of Decimals in MathMasters::sqrt() instead of Hexadecimals representation results in nonequivalence leading to loss of precision.

File: MathMasters

Element: sqrt()

Likelihood: Medium

Financial Impact: High

Severity: High

Details

The MathMasters imlementation is based on the equivalent soladay version. However, the MathMaster implementation uses decimals to represent the constants, while the soladay implementation uses hexadecimals. In Ethereum's Solidity programming language, hexadecimal notation (base-16) is commonly used for specifying literals and values, especially when working with low-level constructs like assembly. Hexadecimal notation is more aligned with the way data is represented in the Ethereum Virtual Machine (EVM). While you can use decimal literals in high-level Solidity code, when

writing assembly code in Solidity, it's generally expected to use hexadecimal notation for representing values and memory locations. In short, the EVM is expecting to receive hexadecimal values and MathMaster sqrt function is submitting decimal values instead. This results in a loss of precision in the MathMasters implementation.

IMPACT

The loss of precision can lead to incorrect results when using the MathMasters implementation.

PROOF OF CONCEPT

Step 1: As a reference implementation refer to solady sqrt: https://github.com/Vectorized/solady/blob/8919f61d14a5e7b32f3d809c9f5fe3ea2ebcbc50/src/utils/FixedPointMathLib.sol#L615

Step 2: Notice that that bottom half of both sqrt implementations (MathMasters and Solady) are identical. Let's place the identical code in a helper function.

```js
function _identicalCodeSqrt(uint256 x, uint256 z) public pure returns (uint256 ret) {
        assembly {
            z := shr(1, add(z, div(x, z)))
            z := shr(1, add(z, div(x, z)))
            z := shr(1, add(z, div(x, z)))
            z := shr(1, add(z, div(x, z)))
            z := shr(1, add(z, div(x, z)))
            z := shr(1, add(z, div(x, z)))
            z := shr(1, add(z, div(x, z)))

            ret := sub(z, lt(div(x, z), z))
        }
    }
```

Step 3: We can then create versions of the two sqrt functions with calls to the identical code.

```js
// The Solady sqrt function with the bottom half replaced with a call to the helper function for the identical code.
function sharedSoladySqrt(uint256 x) public pure returns (uint256 z) {
        assembly {
            z := 181

            let r := shl(7, lt(0xffffffffffffffffffffffffffffffff, x))
            r := or(r, shl(6, lt(0xffffffffffffffff, shr(r, x))))
```

```
            r := or(r, shl(5, lt(0xffffffffff, shr(r, x))))
            r := or(r, shl(4, lt(0xffffff, shr(r, x))))
            z := shl(shr(1, r), z)

            z := shr(18, mul(z, add(shr(r, x), 65536)))
        }
        z = _identicalCodeSqrt(x, z);
    }

// The MathMaster sqrt function with the bottom half replaced with a call to the helper function for the identical code.
    function sharedMathMastersSqrt(uint256 x) internal pure returns (uint256 z) {
        /// @solidity memory-safe-assembly
        assembly {
            z := 181

            let r := shl(7, lt(87112285931760246646623899502532662132735, x))
            r := or(r, shl(6, lt(4722366482869645213695, shr(r, x))))
            r := or(r, shl(5, lt(1099511627775, shr(r, x))))
            // Correct: 16777215 0xffffff
            r := or(r, shl(4, lt(16777002, shr(r, x))))
            z := shl(shr(1, r), z)

            z := shr(18, mul(z, add(shr(r, x), 65536))) // A `mul()` is saved from starting `z` at 181.
        }
        z = _identicalCodeSqrt(x, z);
    }
```

Step 4: We can then conduct a fuzz test to confirm that the two modified sqrt functions along with the helper function are still working correctly.

``` js
// Fuzz test to check that output for both sqrt functions with a shared helper function containing duplicate code appears to be correct.
    function test_fuzz_sharedSqrtFunctions(uint32 fuzzedSolution, uint32 fuzzedRandomNum) public
{
        // ARRANGE: specify input conditions
        uint256 solution = fuzzedSolution;
        vm.assume(solution > 0);
        uint256 randomNum = fuzzedRandomNum;
        uint256 squaredPlusRemainder = solution * solution + (randomNum % solution);

        // ACT: call target contracts
        uint256 mathMastersOuput = Base_Test.sharedMathMastersSqrt(squaredPlusRemainder);
        uint256 soladyOutput = Base_Test.sharedSoladySqrt(squaredPlusRemainder);

        // ASSERT: check output states
        assertEq(solution, mathMastersOuput);
```

```
        assertEq(solution, soladyOutput);
    }
```

Step 5: Once we've verified that the adjusted square root functions are functioning as intended, we can disregard the duplicated code and focus on testing the essential differences in the remaining code for equivalence. To begin, let's create the two square root functions, each containing only their core code.

``` js
// The core Solady function, without the identical code.
    function coreSoladySqrt(uint256 x) public pure returns (uint256 z) {
        assembly {
            z := 181

            let r := shl(7, lt(0xffffffffffffffffffffffffffffffff, x))
            r := or(r, shl(6, lt(0xffffffffffffffff, shr(r, x))))
            r := or(r, shl(5, lt(0xffffffff, shr(r, x))))
            r := or(r, shl(4, lt(0xffff, shr(r, x))))
            z := shl(shr(1, r), z)

            z := shr(18, mul(z, add(shr(r, x), 65536)))
        }
    }

    // The core MathMaster function, without the identical code.
    function coreMathMasterSqrt(uint256 x) internal pure returns (uint256 z) {
        /// @solidity memory-safe-assembly
        assembly {
            z := 181

            // This segment is to get a reasonable initial estimate for the Babylonian method. With a bad
            // start, the correct # of bits increases ~linearly each iteration instead of ~quadratically.
            let r := shl(7, lt(87112285931760246646623899502532662132735, x))
            r := or(r, shl(6, lt(4722366482869645213695, shr(r, x))))
            r := or(r, shl(5, lt(1099511627775, shr(r, x))))
            // Correct: 16777215 0xffffff
            r := or(r, shl(4, lt(16777002, shr(r, x))))
            z := shl(shr(1, r), z)

            // There is no overflow risk here since `y < 2**136` after the first branch above.
            z := shr(18, mul(z, add(shr(r, x), 65536))) // A `mul()` is saved from starting `z` at 181.
        }
    }
```

Step 6: We can now test the two core sqrt functions for equivalence using Halmos.

``` js
```

```
    // halmos returns and error "counterexample-unknown". So we disble the timeout limit for the assertion; thus giving Halmos more time to find a counterexample.
    /// @custom:halmos --solver-timeout-assertion 0

    function check_coreSqrtFunctions() public {
        // ARRANGE: specify input conditions
        uint256 x = svm.createUint256("x");

        // ACT: call target contracts
        uint256 coreMathMastersOuput = Base_Test.coreMathMasterSqrt(x);
        uint256 coreSoladyOutput = Base_Test.coreSoladySqrt(x);

        // ASSERT: check output states
        assertEq(coreMathMastersOuput, coreSoladyOutput);
    }
```

Step 7: Having received a counter example from Halmos, let's test it using Foundry's Forge tool. Run it using "forge test --match-test test_poc_sqrt -vv"

```
  function test_poc_sqrt() public {
        // ARRANGE: specify input conditions
        uint256                                                    x                                                    =
105311293498665291426722909308999732236070323463302251608708546560;

        // ACT: call target contracts
        uint256 mathMastersOuput = Base_Test.coreMathMasterSqrt(x);
        uint256 soladyOutput = Base_Test.coreSoladySqrt(x);
        console.log("mathMastersOuput: ", mathMastersOuput);
        console.log("soladyOutput: ", soladyOutput);

        // ASSERT: check output states
        assertEq(mathMastersOuput, soladyOutput);
    }
```

Tools Used

- Foundry Forge: Fuzz Testing for Correctness
- Halmos: Formal Verification Testing for Equivalence

Recommendations

Use hexadecimals to represent the constants in the MathMasters implementation.

```
- let r := shl(7, lt(87112285931760246646623899502532662132735, x))
- r := or(r, shl(6, lt(4722366482869645213695, shr(r, x))))
- r := or(r, shl(5, lt(1099511627775, shr(r, x))))
- r := or(r, shl(4, lt(16777002, shr(r, x))))

+ let r := shl(7, lt(0xffffffffffffffffffffffffffffffff, x))
+ r := or(r, shl(6, lt(0xffffffffffffffff, shr(r, x))))
+ r := or(r, shl(5, lt(0xffffffff, shr(r, x))))
+ r := or(r, shl(4, lt(0xffffff, shr(r, x))))
```

Medium Severity Findings

N/A

Low Severity Findings

N/A

Informational Level Findings

N/A

CONCLUSION

In conclusion, the smart contract audit has identified two critical issues that demand immediate attention. Firstly, in the MathMasters::mulWadUp() function (H-01), the use of the OR operator in the overflow check, meant to prevent issues in fixed-point arithmetic, ironically introduces a risk of overflow itself. This occurs when conditions involving non-zero values and maximum limits are evaluated. Secondly, in the MathMasters::sqrt() function (H-02), the use of decimal notation for constants instead of the recommended hexadecimal representation creates nonequivalence with the expected values by the Ethereum Virtual Machine (EVM). This mismatch, especially in low-level constructs like assembly, leads to a significant loss of precision in the MathMasters implementation. Addressing these high-severity findings promptly is crucial to ensuring the integrity and functionality of the smart contract.

APPENDICES

NONE