MathMasters Audit Report

Version 1.0

Cyfrin.io

December 13, 2024

MathMasters Audit Report

Dheeraj K

Dec 13, 2024

Prepared by: Dheeraj Kumar Lead Auditor:

- Dheeraj Kumar Yaduwanshi

Table of Contents

Protocol Summary

MathMasters is an arithmetic library for fixed point numbers. It facilitates multiplication and square root operations for numbers. It employs Babylonian method for calculation of square roots.

Disclaimer

Dheeraj Kumar makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: c7643faa1a188a51b2167b68250816f90a9668c6

Scope

```
./src/
# --- MathMasters.sol
```

Roles

XX

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	3
Gas	0
Informational	1
Total	6

Findings

High

[H-1] `MathMasters::sqrt` incorrectly checks the `lt` of a right shift, causing potentially incorrect sqrt values to be returned

Description: The `MathMasters::sqrt` function calculates square root using the Babylonian method. The `MathMasters::sqrt` implementation is similar to that of Solady. However it differs in one way. The Solady implementation uses hexadecimal values and the MathMaster sqrt function uses decimals instead. Due to this, there's a loss of precision as the right values for the hexadecimal numbers are not used.

Impact: The loss of precision can lead to incorrect square root results when using the MathMasters implementation of the Babylonian method.

Proof of Concept: We noticed from Solady and MathMasters that the bottom half of both the implementations are identical.

```
    function sqrt(uint256 x) internal pure returns (uint256 z) {
        /// @solidity memory-safe-assembly
        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)))

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

Let's compare the top half. We can create two functions as below.

```
     function solmateTopHalf(uint256 x) external pure returns (uint256
        z) {
        assembly {
            let y := x

            z := 181
            if iszero(lt(y, 0x100000000000000000000000000000000)) {
                y := shr(128, y)
                z := shl(64, z)
            }
            if iszero(lt(y, 0x10000000000000000)) {
                y := shr(64, y)
                z := shl(32, z)
            }
            if iszero(lt(y, 0x100000000)) {
                y := shr(32, y)
                z := shl(16, z)
            }
            if iszero(lt(y, 0x1000000)) {
                y := shr(16, y)
                z := shl(8, z)
            }
            z := shr(18, mul(z, add(y, 65536)))
        }
    }

    function mathMastersTopHalf(uint256 x) external pure returns (
       uint256 z) {
        assembly {
            z := 181

```

```
30              let r := shl(7, lt
                    (87112285931760246646623899502532662132735, x))
31              r := or(r, shl(6, lt(47223664828696452136 95, shr(r, x))))
32              r := or(r, shl(5, lt(1099511627775, shr(r, x))))
33              r := or(r, shl(4, lt(16777002, shr(r, x))))
34              z := shl(shr(1, r), z)
35
36              z := shr(18, mul(z, add(shr(r, x), 65536)))
37          }
38      }
```

Now, add a rule and run it using Certora that asserts the results obtained from top half of both the implementations.

```
1       rule solmateTopHalfMatchesMathMastersTopHalf(uint256 x) {
2       assert(solmateTopHalf(x) == mathMastersTopHalf(x));
3   }
```

The result of CertoraRun concludes that the output from first half of both the implementations does not match. The value for which the assertion fails is `0xffff2b0000000000000001`.

Let's run a test with the above x value.

```
1       function testCompactFuzz() public {
2           uint256 x = 0xffff2b0000000000000001;
3           assertEq(mathMastersTopHalf(x), solmateTopHalf(x));
4       }
```

The output of running above test is as below:

```
1   2-math-master-audit git:(main) forge test --mt testCompactFuzz -vvvv
2   Compiling...
3   No files changed, compilation skipped
4
5   Ran 1 test for test/MathMasters.t.sol:MathMastersTest
6   [FAIL: assertion failed: 49946489257984 != 49946616774656]
       testCompactFuzz() (gas: 221813)
```

However, the only difference in the implementations is the use of decimal and hexadecimals values. On correctly matching those values by converting them to hex, we realised that `16777002 (or hex: 0xffff2a)` is used instead of `16777215 (or 0xffffff)` and this leads to incorrect results.

Recommended Mitigation: Use hexadecimals instead of decimal values in `MathMasters::sqrt` implementation.

```
1       /// @dev Returns the square root of `x`.
2       function sqrt(uint256 x) internal pure returns (uint256 z) {
3           /// @solidity memory-safe-assembly
```

```
4           assembly {
5               ...
6
7   -           let r := shl(7, lt(87112285931760246646623899502532662132735, x))
8   -           r := or(r, shl(6, lt(4722366482869645213695, shr(r, x))))
9   -           r := or(r, shl(5, lt(1099511627775, shr(r, x))))
10  -           r := or(r, shl(4, lt(16777002, shr(r, x))))
11
12  +           let r := shl(7, lt(0xffffffffffffffffffffffffffffffff, x))
13  +           r := or(r, shl(6, lt(0xffffffffffffffff, shr(r, x))))
14  +           r := or(r, shl(5, lt(0xffffffffff, shr(r, x))))
15  +           r := or(r, shl(4, lt(0xffffff, shr(r, x))))
16
17              ...
18          }
19      }
```

[H-2] `MathMasters::mulWadUp` gives incorrect results for some values of x and y

Description: The `mulWadUp` function calculates `x * y / 1e18` and rounds the result up. There's an `if` statement that increments the value of `x` on certain conditions.

```
1       /// @dev Equivalent to `(x * y) / WAD` rounded up.
2       function mulWadUp(uint256 x, uint256 y) internal pure returns (uint256 z) {
3           /// @solidity memory-safe-assembly
4           assembly {
5               // Equivalent to `require(y == 0 || x <= type(uint256).max / y)`.
6               if mul(y, gt(x, div(not(0), y))) {
7                   mstore(0x40, 0xbac65e5b) // `MathMasters__MulWadFailed()`.
8                   revert(0x1c, 0x04)
9               }
10  @>          if iszero(sub(div(add(z, x), y), 1)) { x := add(x, 1) }
11              z := add(iszero(iszero(mod(mul(x, y), WAD))), div(mul(x, y), WAD))
12          }
13      }
```

The result of the `if` block depends on `iszero(sub(div(add(z, x), y), 1))`. `If` statement will be true for two conditions: - If x and y are equal - If value of x is greater than y such that `x / y < 2`.

For the above two conditions, as `if` block is true, value of `x` will be incremented by 1. However, the final result of (x * y) gives incorrect result due to this increment.

Let's run the following rule using `certora`.

```
    rule check_testMulWadUpFuzz(uint256 x, uint256 y) {
        require (x == 0 || y == 0 || y <= assert_uint256(max_uint256 /
            x));
        uint256 result = mulWadUp(x, y);
        mathint expected = x * y == 0 ? 0 : (x * y - 1) / WAD() + 1;
        assert(result == assert_uint256(expected));
    }
```

Here is the link to Certora Output.

We can see that the assertion fails for below values of x and y. x = 0xde0b6b3a7640000 y = 0xde0b6b3a7640000

Impact: The increment in the value of `x` on above stated conditions gives incorrect result for the multiplication. The results always round up if there's any remainder on product of two values. However, since the value of `x` is already incorrect in certain cases, the result will be incorrect.

Proof of Concept:

Let's run the test for values of x and y obtained from `certora`.

Proof of Code

Add the following test in `MathMasters.t.sol`

```
    function testCheckCertoraOutput() public {
        uint256 x = 0xde0b6b3a7640000;
        uint256 y = 0xde0b6b3a7640000;
        uint256 result = MathMasters.mulWadUp(x, y);
        uint256 expected = (x * y - 1) / 1e18 + 1;
        assertEq(result, expected);
    }
```

The above test fails with the following error:

```
2-math-master-audit git:(main) forge test --mt testCheckCertoraOutput -
    vvvv
Compiling...
No files changed, compilation skipped

Ran 1 test for test/MathMasters.t.sol:MathMastersTest
[FAIL: assertion failed: 1000000000000000001 != 1000000000000000000]
    testCheckCertoraOutput() (gas: 3622)
```

The actual result for `x * y / 1e18` should be `1000000000000000000`, but since x and y are equal, the `if` condition is `true` and hence, the value of x is incremented by 1. Thus, the result is an incorrect value `1000000000000000001`

Recommended Mitigation: Make the following changes in the `mulWadUp` function.

```
    /// @dev Equivalent to `(x * y) / WAD` rounded up.
    function mulWadUp(uint256 x, uint256 y) internal pure returns (uint256 z) {
        /// @solidity memory-safe-assembly
        assembly {
            // Equivalent to `require(y == 0 || x <= type(uint256).max / y)`.
            if mul(y, gt(x, div(not(0), y))) {
                mstore(0x40, 0xbac65e5b) // `MathMasters__MulWadFailed()`.
                revert(0x1c, 0x04)
            }
-           if iszero(sub(div(add(z, x), y), 1)) { x := add(x, 1) }
            z := add(iszero(iszero(mod(mul(x, y), WAD))), div(mul(x, y), WAD))
        }
    }
```

Low

[L-1] Solidity version 0.8.3 does not support custom errors, breaks compilation

Description: Custom errors in Solidity were introduced in version `0.8.4`. Previous versions does not support custom errors and would likely fail during compilation.

Impact: The compilation would break.

Recommended Mitigation: Use of Solidity version greater than or equal to `0.8.4` is suggested.

[L-2] In `MathMasters::mulWad` and `MathMasters::mulWadUp`, the revert reason is empty

Description: The functions `MathMasters::mulWad` and `MathMasters::mulWadUp` in case of an error would revert with the following reason.

```
    revert(0x1c, 0x04);
```

However, nothing is defined at that location in the memory. Thus, the revert message would be blank.

[L-3] Wrong function selector for `MathMasters::MathMasters__FullMulDivFailed()` custom error

Description: The function selector for `MathMasters__FullMulDivFailed()` is `0x41672c55`, but `0xbac65e5b` is used instead.

Informational

[I-1] Custom error function selector is written at Free Memory Pointer's position

Description: In Solidity, the offset `0x40` is special and reserved for free memory pointer. It keeps track of next available memory. In `MathMasters::mulWad` and `MathMasters::mulWadUp`, custom error code is overwritten at the position of free memory pointer.