Math Masters Audit Report

Version 1.0

Prakash Yadav

September 17, 2025

Math Masters Audit Report

Prakash Yadav

September 17, 2025

Prepared by: Prakash Yadav Lead Auditors: - Prakash Yadav

Table of Contents

Protocol Summary

Disclaimer

Prakash Yadav 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

- In Scope:

```
#-- MathMasters.sol
```

Roles

XX

Executive Summary

Issues found

Category	No. of Issues
High	2
Medium	0
Low	3
Total	5

Findings

High

[H-1]: In `MathMasters::mulWadUp` function is intended to perform fixed-point multiplication with rounding up.

Description: In `MathMasters:mulWadUp` function calculates the product of x and y through mul(x,y). But multiplication can overflow if x and y are large enough,causing incorrect results or revert or may be produced incorrect results. If the rounding logic is misunderstood, it can lead to off-by-one errors in mathematical calculations.

```
1            function mulWadUp(uint256 x, uint256 y) internal pure returns
                 (uint256 z) {
2          /// @solidity memory-safe-assembly
3          assembly {
4              // Equivalent to `require(y == 0 || x <= type(uint256).max
                   / y)`.
5              if mul(y, gt(x, div(not(0), y))) {
6                  mstore(0x40, 0xbac65e5b) // `MathMasters__MulWadFailed
                       ()`.
7                  revert(0x1c, 0x04)
```

```
8               }
9   @>          if iszero(sub(div(add(z, x), y), 1)) { x := add(x, 1) }
10              z := add(iszero(iszero(mod(mul(x, y), WAD))), div(mul(x, y)
                    , WAD))
11          }
12      }
```

Impact: 1. If overflow occurs, the function will revert or produce incorrect results. 2. If the rounding logic is misunderstood or misapplied, it can lead to off-by-one errors in calculations, affecting mathematical correctness.

Proof of Concepts:

PoC

Halmos:

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

```
1  function check_testMulWadUpFuzz(uint256 x, uint256 y) public pure {
2          if (x == 0 || y == 0 || y <= type(uint256).max / x) {
3              uint256 result = MathMasters.mulWadUp(x, y);
4              uint256 expected = x * y == 0 ? 0 : (x * y - 1) / 1e18 + 1;
5              assert(result== expected);
6          }
7      }
```

Certora:

Create a file `CompactCodeBase.sol` in test folder.

```
1  // SPDX-License-Identifier: AGPL-3.0-only
2  pragma solidity ^0.8.3;
3
4  import {MathMasters} from "../src/MathMasters.sol";
5
6
7
8  contract CompactCodeBase{
9      function mulWadUp_public(uint256 x,uint256 y) external pure returns
           (uint256) {
10         return MathMasters.mulWadUp(x,y);
11     }
12 }
```

Also, Create a folder certora and create two file.

1. MulWadUp.conf

```
1  {
2      "files":[
```

```
        "./src/MathMasters.sol",
        "./certora/MathMastersHarness.sol"
    ],
    "verify":"MathMastersHarness:./certora/MulWadUp.spec",
    "wait_for_results":"all",
    "rule_sanity":"basic",
    "optimistic_loop":true,
    "msg":"Verification of MulWad"
}
```

2. MulWadUp.spec

```
/*
* Verification of mulWadUp for MathMasters
*/


methods{
    function mulWadUp_public(uint256 x,uint256 y) external returns (uint256) envfree;
}

definition WAD() returns uint256=1000000000000000000;

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

Recommended Mitigation: In `MathMaster::mulWadUp` function ,remove the line which caused rounding logic in it.

```
     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))
```

```
11          }
12      }
```

[H-2]: In `MathMasters::sqrt` function it used incorrect value which caused error in sqrt function.

Description: `MathMasters::sqrt` function used incorrect values which could return incorrect value in calculating and getting value of sqrt function.

```
1         function sqrt(uint256 x) internal pure returns (uint256 z) {
2          /// @solidity memory-safe-assembly
3          assembly {
4              z := 181
5
6              // This segment is to get a reasonable initial estimate for
                   the Babylonian method. With a bad
7              // start, the correct # of bits increases ~linearly each
                   iteration instead of ~quadratically.
8              let r := shl(7, lt
                   (87112285931760246646623899502532662132735, x))
9              r := or(r, shl(6, lt(4722366482869645213695, shr(r, x))))
10             r := or(r, shl(5, lt(1099511627775, shr(r, x))))
11             // Correct: 16777215 0xffffff
12 @>           r := or(r, shl(4, lt(16777002, shr(r, x))))
13             z := shl(shr(1, r), z)
14
15             // There is no overflow risk here since `y < 2**136` after
                   the first branch above.
16             z := shr(18, mul(z, add(shr(r, x), 65536))) // A `mul()` is
                    saved from starting `z` at 181.
17
18             // Given the worst case multiplicative error of 2.84 above,
                    7 iterations should be enough.
19             z := shr(1, add(z, div(x, z)))
20             z := shr(1, add(z, div(x, z)))
21             z := shr(1, add(z, div(x, z)))
22             z := shr(1, add(z, div(x, z)))
23             z := shr(1, add(z, div(x, z)))
24             z := shr(1, add(z, div(x, z)))
25             z := shr(1, add(z, div(x, z)))
26
27             // If `x+1` is a perfect square, the Babylonian method
                   cycles between
28             // `floor(sqrt(x))` and `ceil(sqrt(x))`. This statement
                   ensures we return floor.
29             // See: https://en.wikipedia.org/wiki/Integer_square_root#
                   Using_only_integer_division
30             z := sub(z, lt(div(x, z), z))
```

```
31            }
32        }
```

Impact:

1. Incorrect numeric results: the wrong comparison constant causes the initial estimate for the integer square root to be off for a range of inputs. That shifts the Babylonian iterations' starting point and can lead to a wrong final floor(sqrt(x)) value for some inputs.
2. Downstream computation errors: any users of `sqrt` (or of helpers that rely on the same top-half estimate) will observe incorrect derived values (e.g., bounds, normalized magnitudes, or other integer-math results), which can silently corrupt higher-level math operations.
3. Deterministic but subtle: this is a data-only (non-exploitable) correctness issue — it does not introduce a reentrancy or memory-safety vulnerability, but it can cause deterministic calculation deviations that are hard to detect without test coverage.
4. Auditability and client trust: consumers that assume correct integer-square-root behavior (for proofs, invariants, or financial calculations) may produce incorrect proofs or balances; this raises integrity concerns and requires either a patch or explicit documentation of the limitation.

Proof of Concepts:

PoC

Halmos:

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

```
1         import {CompactCodeBase} from "./CompactCodeBase.sol";
2         .
3         .
4         .
5          function testCompactFuzz(uint256 x) public{
6           CompactCodeBase cc=new CompactCodeBase();
7           assertEq(cc.mathMastersTopHalf(x),cc.solmateTopHalf(x));
8       }
9
10      function testCompactFuzz() public{
11          uint256 x=0xffffff000000000000000000000000;
12          CompactCodeBase cc=new CompactCodeBase();
13          assertEq(cc.mathMastersTopHalf(x),cc.solmateTopHalf(x));
14      }
```

Certora:

Create a file `CompactCodeBase.sol` in test folder.

```
1
2  pragma solidity ^0.8.3;
3
```

```
import {MathMasters} from "../src/MathMasters.sol";



contract CompactCodeBase{

    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
            let r := shl(7, lt
                (87112285931760246646623899502532662132735, x))
            r := or(r, shl(6, lt(47223664828696452136 95, shr(r, x))))
            r := or(r, shl(5, lt(1099511627775, shr(r, x))))
            r := or(r, shl(4, lt(16777215, shr(r, x))))
            z := shl(shr(1, r), z)
            z := shr(18, mul(z, add(shr(r, x), 65536)))
        }
    }

}
```

Also, Create a folder certora and create two file.

1. Sqrt.conf

```
    /*
* Verification of sqrt for MathMasters
*/

methods{
    function mathMastersTopHalf(uint256) external returns uint256
       envfree;
    function solmateTopHalf(uint256) external returns uint256 envfree;
}

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

2.Sqrt.spec

```
   {
    "files":[
        "./src/MathMasters.sol",
        "./test/CompactCodeBase.sol"
    ],
    "verify":"CompactCodeBase:./certora/sqrt.spec",
    "wait_for_results":"all",
    "rule_sanity":"basic",
    "optimistic_loop":true,
    "msg":"Verification of Sqrt"
}
```

Recommended Mitigation: After converting each of value to hex it provided incorrect value in line `r := or(r, shl(4, lt(16777002, shr(r, x))))` and it should gives us output as `0xffff2a` after put input as `cast --to-base 16777002 hex`.

```
function sqrt(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))))
-           r := or(r, shl(4, lt(16777002, shr(r, x))))
+           r := or(r, shl(4, lt(16777215, shr(r, x))))
            z := shl(shr(1, r), z)
```

```
14
15              // There is no overflow risk here since `y < 2**136` after
                    the first branch above.
16              z := shr(18, mul(z, add(shr(r, x), 65536))) // A `mul()` is
                     saved from starting `z` at 181.
17
18              // Given the worst case multiplicative error of 2.84 above,
                     7 iterations should be enough.
19              z := shr(1, add(z, div(x, z)))
20              z := shr(1, add(z, div(x, z)))
21              z := shr(1, add(z, div(x, z)))
22              z := shr(1, add(z, div(x, z)))
23              z := shr(1, add(z, div(x, z)))
24              z := shr(1, add(z, div(x, z)))
25              z := shr(1, add(z, div(x, z)))
26
27              // If `x+1` is a perfect square, the Babylonian method
                    cycles between
28              // `floor(sqrt(x))` and `ceil(sqrt(x))`. This statement
                    ensures we return floor.
29              // See: https://en.wikipedia.org/wiki/Integer_square_root#
                    Using_only_integer_division
30              z := sub(z, lt(div(x, z), z))
31          }
32      }
```

Medium

Low

[L-1]: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/MathMasters.sol Line: 3

```
1  pragma solidity ^0.8.3;
```

[L-2]: PUSH0 is not supported by all chains

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in

case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

- Found in src/MathMasters.sol Line: 3

```
1 pragma solidity ^0.8.3;
```

L-3: Unused Custom Error

it is recommended that the definition be removed when custom error is unused.

- Found in src/MathMasters.sol Line: 14

```
1     error MathMasters__FactorialOverflow();
```

- Found in src/MathMasters.sol Line: 15

```
1     error MathMasters__MulWadFailed();
```

- Found in src/MathMasters.sol Line: 16

```
1     error MathMasters__DivWadFailed();
```

- Found in src/MathMasters.sol Line: 17

```
1     error MathMasters__FullMulDivFailed();
```

Informational

Gas