MathMasters Audit Report

SnavOhBurmaa

31, 5, 2026

1 MathMasters Audit Report

Prepared by: SnavOhBurmaa Lead Auditors:

- SnavOhBurmaa

Assisting Auditors:

- None

2 Table of contents

See table

3 About Me

Hi, I'm Khant Wai Yan Aung (SnavOhBurmaa). I'm currently focusing on smart contract security audit. This is one report in my ongoing audit series. Thanks for reading.

4 Risk Classification

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

5 Audit Details

Source repository: https://github.com/Cyfrin/2-math-master-audit

The findings described in this document correspond the following commit hash:

`a7ac04531b7868ae2c23281400b4b1819dbc6c38`

5.1 Scope

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

The library file `src/MathMasters.sol` is the only in-scope contract. It exposes 3 internal pure functions: `mulWad`, `mulWadUp`, `sqrt`, and 4 custom errors.

6 Protocol Summary

MathMasters is a small fixed-point math library inspired by Solady and Solmate. It provides three operations:

- `mulWad(x, y)` — `(x * y) / 1e18` rounded down
- `mulWadUp(x, y)` — `(x * y) / 1e18` rounded up
- `sqrt(x)` — integer square root (floor)

The library is intended to be used by DeFi protocols that need 18-decimal fixed-point math (AMM curves, vault share math, collateral valuation, etc.).

6.1 Roles

- **Library consumer:** Any contract that imports `MathMasters` and calls one of its 3 functions. The library is `internal pure`, so callers inline the bytecode into their own contracts.

7 Findings

7.1 High

7.1.1 [H-1] `MathMasters :: mulWadUp` wrong ceiling, error grows with input magnitude

Description: Line 57 of `MathMasters.sol` does this:

```
if iszero(sub(div(add(z, x), y), 1)) { x := add(x, 1) }
```

`z` is the return variable. In Yul, return variables start at `0`. So `add(z, x)` is just `x`. The line becomes:

> "If `x / y == 1`, then change `x` to `x + 1`."

This is wrong logic that was added on top of the otherwise-correct Solady code. It silently bumps `x` whenever `y <= x < 2y`, and the final ceiling answer is computed on the wrong `x`.

Risk:

- **Impact: HIGH** — wrong ceiling rounding result. The error is **not bounded to 1 wei**; it scales with input magnitude:

Input	`mulWadUp` returns	Correct ceiling	Off by
`mulWadUp(1e18, 1e18 - 1)`	`1e18`	`1e18 - 1`	1
`mulWadUp(1e18, 1e18)`	`1e18 + 1`	`1e18`	1
`mulWadUp(2e18, 2e18)`	`4e18 + 2`	`4e18`	2
`mulWadUp(2.5e18, 2.5e18)`	`6.25e18 + 3`	`6.25e18`	3
`mulWadUp(1e30, 1e30)`	`~1e42 + 1e12`	`1e42`	~1e12

For DeFi protocols using this: vault share price drift, fee under/overcharges that scale with TVL, broken “deposit then withdraw” invariants, and a widened inflation-attack surface.

- **Likelihood: HIGH** — any caller passing `(x, y)` with `y <= x < 2y` (extremely common range) triggers it. The existing fuzz test finds a counterexample within ~1200 runs, and the stateful invariant fuzz finds one within 5 sequences.

Proof of Concept: Add this test

```
function testMulWadUpOffByOne() public {
    uint256 x = 1e18;
    uint256 y = 1e18 - 1;

    uint256 result = MathMasters.mulWadUp(x, y);
    uint256 expected = (x * y == 0) ? 0 : (x * y - 1) / 1e18 + 1;

    console2.log("result  ", result);
    console2.log("expected", expected);

    assertEq(result, expected); // FAILS: result == 1e18, expected
== 1e18 - 1
}
```

Run `forge test --match-test testMulWadUpOffByOne -vvv`. You will see:

```
result   1000000000000000000
expected 999999999999999999
Error: a == b not satisfied [uint]
      Left: 1000000000000000000
     Right: 999999999999999999
```

Recommended Mitigation: Delete line 57 completely. The function should match the clean Solady form

```
function mulWadUp(uint256 x, uint256 y) internal pure returns
        (uint256 z) {
    /// @solidity memory-safe-assembly
    assembly {
        if mul(y, gt(x, div(not(0), y))) {
-           mstore(0x40, 0xbac65e5b) //
        `MathMasters__MulWadFailed()`.
+           mstore(0x00, 0xa56044f7) // correct
        `MathMasters__MulWadFailed()` selector
            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))
    }
}
```

7.1.2 [H-2] `MathMasters :: sqrt` incorrectly checks the `lt` of a right shift, causing potentially incorrect sqrt values to be returned

Description: The following is a snippet from the `MathMasters :: sqrt` function:

```
// 871122859317602466466238995025326621327 35 ==
        0xffffffffffffffffffffffffffffffffff
let r := shl(7, lt(871122859317602466466238995025326621327 35, x))
// 4722366482869645213695 == 0xffffffffffffffffff
r := or(r, shl(6, lt(4722366482869645213695, shr(r, x))))
// 1099511627775 == 0xffffffffff
r := or(r, shl(5, lt(1099511627775, shr(r, x))))
// @audit this should be 16777215 / 0xffffff
// Right now, it's 0xffff2a!
@>  r := or(r, shl(4, lt(16777002, shr(r, x))))
```

Here you can see multiple `lt` and `shl` operations being used to calculate the initial estimate for the square root. The math involved uses the Babylonian method for computing the integer square root. The first three constants in the chain are correctly `2^136 - 1`, `2^72 - 1`, and `2^40 - 1`. The fourth one is supposed to be `2^24 - 1` (`16777215`, `0xffffff`), but the code uses `16777002` (`0xffff2a`). The author's own comment on the previous line even reads "Correct: 16777215 0xffffff" — confirming the off-by-213 typo.

When the wrong constant fires (or fails to fire) for inputs near the 24-bit boundary, `r` receives the wrong shift amount, the initial estimate `z` is off, and the 7 Babylonian iterations start from the wrong place. The math is, by construction, no longer guaranteed to converge to the correct floor square root.

Risk:

- **Impact: HIGH** — a math primitive in a fixed-point library is broken. Any consumer doing AMM curve math (e.g. constant-product reserves → liquidity), vault share math, collateral valuation, or general DeFi math relying on `sqrt` correctness can mis-mint, mis-burn, or mispriced their internal accounting. A library that is sold as "Solady / Solmate inspired" raising wrong sqrt values is a critical correctness failure.
- **Likelihood: HIGH** — the wrong constant is present in every compiled artifact and runs on every single call to `sqrt`. The bug is not gated by a rare input or external call. The only reason no exploit triggers in the current foundry/Certora suites is that the 7 Babylonian iterations happen to absorb the bad initial estimate — a coincidence, not a guarantee. Anyone reducing the iteration count, refactoring the estimate path, or porting this code into a different scaling regime trips the latent bug instantly.

Proof of Concept: The boundary test below documents the affected range. It passes today only because the iteration count is generous enough to hide the bug:

```
function testSqrtBoundaryNearTwoPow24() public view {
    for (uint256 x = 16777002 - 5; x <= 16777215 + 5; x++) {
        assert(MathMasters.sqrt(x) == solmateSqrt(x));
    }
}
```

Differential fuzz against Solmate / Uniswap also still passes at 500k runs:

```
forge test --match-test testSqrtFuzzSolmate --fuzz-runs 500000
[PASS] testSqrtFuzzSolmate(uint256) (runs: 500000, ...)
```

Recommended Mitigation: Use the correct constant `16777215` (`0xffffff`, `2^24 - 1`) and document the derivation alongside the other three constants:

```
- // 87112285931760246646623899502532662132735 ==
        0xffffffffffffffffffffffffffffffffffffff
+ // 2^136 - 1 == 87112285931760246646623899502532662132735 ==
        0xffffffffffffffffffffffffffffffffffffff
  let r := shl(7, lt(87112285931760246646623899502532662132735,
        x))
- // 4722366482869645213695 == 0xffffffffffffffffff
+ // 2^72  - 1 == 4722366482869645213695 == 0xffffffffffffffffff
  r := or(r, shl(6, lt(4722366482869645213695, shr(r, x))))
- // 1099511627775 == 0xffffffffff
+ // 2^40  - 1 == 1099511627775 == 0xffffffffff
  r := or(r, shl(5, lt(1099511627775, shr(r, x))))
- // Correct: 16777215 0xffffff
- r := or(r, shl(4, lt(16777002, shr(r, x))))
+ // 2^24  - 1 == 16777215 == 0xffffff
+ r := or(r, shl(4, lt(16777215, shr(r, x))))
```

7.2 Medium

7.2.1 [M-1] Wrong error selector AND wrong memory location in `mulWad` and `mulWadUp` revert path

Description: Both functions revert like this:

```
mstore(0x40, 0xbac65e5b) // `MathMasters__MulWadFailed()`.
revert(0x1c, 0x04)
```

Two bugs at once:

1. **Wrong selector.** The number `0xbac65e5b` is the selector for **Solady's** error `MulWadFailed()` — not for `MathMasters__MulWadFailed()`. Verified with `cast keccak`:

```
cast keccak "MathMasters__MulWadFailed()" -> 0xa56044f7...
cast keccak "MulWadFailed()"              -> 0xbac65e5b...
```

1. **Wrong memory location.** `0x40` is the slot that holds the Solidity **free memory pointer**. Writing the selector there does not place it where `revert(0x1c, 0x04)` reads from. The revert reads bytes `0x1c..0x20` (the last 4 bytes of slot `0x00`), which were never written to. So the actual error data returned is `0x00000000`.

Risk:

- **Impact: MEDIUM** — callers cannot programmatically tell which error happened. Error decoders, integration tooling, and on-chain `try/catch` flows all break, reports `MathMasters__MulWadFailed` as unused because the AST scanner cannot see any code raising it. The function still reverts so funds are not directly at risk, but failure handling is broken.
- **Likelihood: HIGH** — every single revert from `mulWad` / `mulWadUp` returns the wrong (zero) error data. 100% reproducible.

Proof of Concept: Add this test

```
function testMulWadRevertDataIsZero() public {
    try this.callMulWadOverflow(type(uint256).max, 2) {
        revert("expected revert");
    } catch (bytes memory data) {
        console2.logBytes(data);
        assertEq(data.length, 4);
        assertEq(bytes4(data),
MathMasters.MathMasters__MulWadFailed.selector); // FAILS
    }
}

function callMulWadOverflow(uint256 x, uint256 y) external pure
returns (uint256) {
    return MathMasters.mulWad(x, y);
}
```

Run `forge test --match-test testMulWadRevertDataIsZero -vvv`. You will see:

```
0x00000000
Error: a == b not satisfied [bytes32]
      Left:
0x0000000000000000000000000000000000000000000000000000000000000000
     Right:
0xa56044f700000000000000000000000000000000000000000000000000000000
```

Recommended Mitigation: Write the **correct** selector to slot `0x00`, then revert from `0x1c`:

```
- mstore(0x40, 0xbac65e5b)
+ mstore(0x00, 0xa56044f7) // selector for
        MathMasters__MulWadFailed()
  revert(0x1c, 0x04)
```

Apply the same fix in `mulWadUp`.

7.2.2 [M-2] Custom errors declared but never used; library API is incomplete

Description: The library declares 4 custom errors (lines 14–17), but only one (`MathMasters__MulWadFailed`) is referenced — and even that one is referenced with the wrong selector. The other 3 errors point to functions (`factorial`, `divWad`, `fullMulDiv`) that do not exist in the library at all.

Risk:

- **Impact: MEDIUM** — integrators reading the ABI may assume the corresponding functions exist and write code that never compiles or always reverts. Misleading library surface, plus wasted contract bytecode.
- **Likelihood: MEDIUM** — anyone reading the ABI to integrate will see the 3 dead errors, all four as unused, confirming the dead-code state at the AST level.

Proof of Concept: Run Aderyn:

```
$ make aderyn
L-2: Unused Error
- error MathMasters__FactorialOverflow();
- error MathMasters__MulWadFailed();      // unused because of M-1
- error MathMasters__DivWadFailed();
- error MathMasters__FullMulDivFailed();
```

Recommended Mitigation: Either implement the missing functions (`factorial`, `divWad`, `fullMulDiv`) or remove the unused error declarations.

```
- error MathMasters__FactorialOverflow();
  error MathMasters__MulWadFailed();
- error MathMasters__DivWadFailed();
- error MathMasters__FullMulDivFailed();
```

7.3 Low

7.3.1 [L-1] Floating pragma `^0.8.3` allows old, buggy compiler versions

Description: `pragma solidity ^0.8.3;` means any 0.8.x with x ≥ 3 will compile this library. For a math library that may be inlined into many production contracts, lock to a specific, audited compiler.

Risk:

- **Impact: LOW** — opens the door to compiler-level bugs that may or may not affect the assembly blocks.
- **Likelihood: MEDIUM** — depends on the consumer's compiler choice. Most modern projects pin a recent version, but an L2-targeted build using an old 0.8.x is plausible.

Recommended Mitigation:

```
- pragma solidity ^0.8.3;
+ pragma solidity 0.8.20;
```

7.3.2 [L-2] Insufficient test coverage on `sqrt`

Description: `testSqrt()` in `MathMasters.t.sol` only checks 6 hard-coded inputs. It never tests near the bit-length boundaries (`2^24`, `2^40`, `2^72`, `2^136`). The fuzz tests at lines 54–60 are good but run only 1000 times by default — they should run much more in CI.

Risk:

- **Impact: LOW** — no live bug today; coverage hygiene only.
- **Likelihood: MEDIUM** — future modifications (reducing iteration count, changing the estimate path) can regress silently with no test catching it.

7.3.3 [L-3] `mulWad` / `mulWadUp` revert from assembly without bubbling structured error data

Description: Even after M-1 is fixed and the correct selector is returned, callers should know that this library reverts via assembly. There is no NatSpec documenting the revert behavior.

Recommended Mitigation: Add NatSpec `@custom:reverts` tag to each function.

7.4 Gas

7.4.1 [G-1] `mulWadUp` recomputes `mul(x, y)` twice on line 58. Cache the product.

```
function mulWadUp(uint256 x, uint256 y) internal pure returns
        (uint256 z) {
    /// @solidity memory-safe-assembly
    assembly {
        if mul(y, gt(x, div(not(0), y))) {
            mstore(0x00, 0xa56044f7)
            revert(0x1c, 0x04)
        }
-       z := add(iszero(iszero(mod(mul(x, y), WAD))), div(mul(x,
        y), WAD))
+       let p := mul(x, y)
+       z := add(iszero(iszero(mod(p, WAD))), div(p, WAD))
    }
}
```

7.4.2 [G-2] Dead error declarations cost contract bytecode size

Three custom errors declared and never used. Remove them (see M-2).

7.5 Info

7.5.1 [I-1] Magic numbers in `sqrt` have no named constants or derivation comments

Description: Lines 74–78 use 4 huge hex/decimal constants. Without context, future readers cannot easily check them. This is exactly how the H-2 wrong-constant slipped through.

Recommended Mitigation: Add comments showing the derivation, or use local `let` bindings with descriptive names.

7.5.2 [I-2] Stale comment on line 81 references a variable `y` that does not exist

Description: The comment says "since `y < 2**136`" but the function does not have a local named `y`. This is a leftover from porting Solady's code. Confusing for readers and auditors.

Recommended Mitigation: Update the comment to match the actual variable names.