MathMasters Audit Report

Sagar Rana

August 9, 2025

MathMasters Audit Report

Sagar Rana

August 9, 2025

Prepared by: Sagar Rana

Table of Contents

- [L-01] Use of Custom Errors in Solidity 0.8.3 (Unsupported Feature Causes Revert Failures)
- [L-02] Incorrect Error Selector Used in Assembly Revert (Invalid Revert Signature)
- [L-03] Incorrect Error Selector Used in Assembly Revert (Invalid Revert Signature)
- [L-04] Incorrect Constant in `sqrt` Initial Estimate Calculation (Potential Edge Case Inaccuracy)

Protocol Summary

Gas optimized math library for fixed-point operations. Uses inline assembly for calculations.

Disclaimer

Sagar Rana 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

Scope

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

Commit Hash

c7643faa1a188a51b2167b68250816f90a9668c6

Roles

- none

Issues found

Severity	No.of issues
High	2
Medium	1
Low	4
Total	7

Findings

High

[H-01] Overwriting Free Memory Pointer at 0x40 (Memory Corruption Risk)

Description:
In the `mulWad` function, inline assembly is used for gas-efficient arithmetic and custom error handling. However, the following line mistakenly overwrites the free memory pointer stored at memory slot `0x40`, which is reserved by Solidity for tracking the next available free memory:

```
function mulWad(uint256 x, uint256 y) internal pure returns (uint256 z)
    {
    // @solidity memory-safe-assembly
    assembly {
        // 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           z := div(mul(x, y), WAD)
10      }
11  }
```

Slot `0x40` holds the free memory pointer, and writing to it directly corrupts this tracking, which can lead to unpredictable behavior for any subsequent memory operations that rely on Solidity's allocator. This violates Solidity's memory management conventions and can result in bugs that are difficult to diagnose.

Impact:
Overwriting the free memory pointer can lead to: - Memory corruption. - Clobbering of variables or data structures. - Breakage in higher-level Solidity code that allocates memory after this call. - Increased difficulty in debugging and maintaining the contract due to non-standard memory usage.

Recommended Mitigation:
Do not use memory slot `0x40` for storing custom data. Instead, store the error selector at a scratch space like `0x00`, and revert with it properly:

```
1  -mstore(0x40, 0xbac65e5b)
2  +mstore(0x1c, 0xa56044f7)  // matches with the revert call which
       returns from 0x1c address in next line
3  revert(0x1c, 0x04)
```

[H-02] Overwriting Free Memory Pointer at 0x40 (Memory Corruption Risk)

Description:
In the `mulWadUp` function, inline assembly is used for gas-efficient arithmetic and custom error handling. However, the following line mistakenly overwrites the free memory pointer stored at memory slot `0x40`, which is reserved by Solidity for tracking the next available free memory:

```
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  }
```

Slot `0x40` holds the free memory pointer, and writing to it directly corrupts this tracking, which can lead to unpredictable behavior for any subsequent memory operations that rely on Solidity's allocator. This violates Solidity's memory management conventions and can result in bugs that are difficult to diagnose.

Impact:
Overwriting the free memory pointer can lead to: - Memory corruption. - Clobbering of variables or data structures. - Breakage in higher-level Solidity code that allocates memory after this call. - Increased difficulty in debugging and maintaining the contract due to non-standard memory usage.

Recommended Mitigation:
Do not use memory slot `0x40` for storing custom data. Instead, store the error selector at a scratch space like `0x00`, and revert with it properly:

```
1  -mstore(0x40, 0xbac65e5b)
2  +mstore(0x1c, 0xa56044f7)  // matches with the revert call which
       returns from 0x1c address in next line
3  revert(0x1c, 0x04)
```

Medium

[M-01] Incorrect Rounding Logic in `mulWadUp` Leads to Precision Errors in Edge Cases

Description:
The `mulWadUp` function aims to perform fixed-point multiplication with rounding up. However, the rounding logic implemented in this line is flawed:

```
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          }
```

```
@>          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))
        }
    }
```

@> This line intends to adjust `x` upward to ensure the result rounds up when there's a remainder. But it introduces incorrect behavior due to poor assumptions about the intermediate values of `x`, `y`, and `z`, especially when `z` is zero or very small. The flawed calculation causes the final result to be lower than expected — violating the "round up" guarantee.

These precision errors only manifest in **rare edge cases** that are difficult to catch through fuzzing due to specific alignment of operands.

Impact:
For certain rare inputs, the function **returns a value that is one less than the expected result**, breaking the invariant that the function should always round up when there's a remainder. This undermines trust in the library's mathematical guarantees and can silently affect protocols relying on exact token amounts or financial calculations (e.g., fees, shares, interest accrual).

Proof of Concept: Ran this with `forge test --match-test test_mulWadUpCases`:

```
function test_mulWadUpCases() public {
    uint256 x = 0xde0b6b3a7640001; // 1e18 + 1
    uint256 y = 0xde0b6b3a7640000; // 1e18
    uint256 result = MathMasters.mulWadUp(x, y);
    uint256 expected = x * y == 0 ? 0 : (x * y - 1) / 1e18 + 1;
    assertEq(result, expected);
}
```

Recommended Mitigation:
Remove this line:

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

This ensures correctness without relying on indirect or error-prone arithmetic tricks.

Low

[L-01] Use of Custom Errors in Solidity 0.8.3 (Unsupported Feature Causes Revert Failures)

Description:
The `MathMasters` library defines custom errors (e.g., `MathMasters__FactorialOverflow`, `MathMasters__MulWadFailed`, etc.), which are only supported starting from Solidity version

0.8.4. However, the pragma version specified is `^0.8.3`, which allows compilation with Solidity 0.8.3, a version that does not support custom errors. Attempting to use these custom errors in a compiler version that does not recognize them can lead to compilation failure or unintended behavior during runtime.

Impact:
If compiled with Solidity 0.8.3, any call that attempts to revert using these custom errors will fail to compile. This breaks the intended error handling mechanism and may result in silent failures or incorrect fallback behavior. Moreover, developers using this library expecting structured error messages may encounter ambiguous or misleading errors, which hinders debugging and security auditing.

Recommended Mitigation: Update the pragma version to ^0.8.4 or higher to ensure compatibility with custom errors:

```
-pragma solidity ^0.8.3;
+pragma solidity ^0.8.4;
```

[L-02] Incorrect Error Selector Used in Assembly Revert (Invalid Revert Signature)

Description:
The `mulWad` function attempts to revert using a custom error (`MathMasters__MulWadFailed()`) by manually storing its selector in memory using inline assembly. However, the error selector used (`0xbac65e5b`) is incorrect. The correct selector for `MathMasters__MulWadFailed()` is `0xa56044f7`, as verified via `cast sig "MathMasters__MulWadFailed()"`. This mismatch causes the revert to emit an invalid error signature, which cannot be decoded by external tooling or contracts expecting the correct error format.

```
function mulWad(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)
        }
        z := div(mul(x, y), WAD)
    }
}
```

Impact:
Incorrect error selectors result in revert messages that do not correspond to any declared custom

error. This breaks standard decoding and logging mechanisms, reducing the usefulness of the revert for debugging or off-chain analysis. It also causes inconsistencies between expected and actual behavior when catching errors via interfaces or tools like Hardhat/Foundry.

Proof of Concept:

```
function mulWad(uint256 x, uint256 y) internal pure returns (uint256 z)
    {
    assembly {
        if mul(y, gt(x, div(not(0), y))) {
            // Incorrect error selector
            mstore(0x40, 0xbac65e5b)
            revert(0x1c, 0x04)
        }
        z := div(mul(x, y), WAD)
    }
}
```

Comparing this to the correct selector:

```
cast sig "MathMasters__MulWadFailed()"
# => 0xa56044f7
```

Recommended Mitigation: use the correct function selector:

```
-mstore(0x40, 0xbac65e5b)
+mstore(0x40, 0xa56044f7)
```

[L-03] Incorrect Error Selector Used in Assembly Revert (Invalid Revert Signature)

Description:

The `mulWadUp` function attempts to revert using a custom error (`MathMasters__mulWadFailed()`) by manually storing its selector in memory using inline assembly. However, the error selector used (`0xbac65e5b`) is incorrect. The correct selector for `MathMasters__mulWadFailed()` is `0xa56044f7`, as verified via `cast sig "MathMasters__mulWadFailed()"`. This mismatch causes the revert to emit an invalid error signature, which cannot be decoded by external tooling or contracts expecting the correct error format.

```
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))
    }
}
```

Impact:
Incorrect error selectors result in revert messages that do not correspond to any declared custom error. This breaks standard decoding and logging mechanisms, reducing the usefulness of the revert for debugging or off-chain analysis. It also causes inconsistencies between expected and actual behavior when catching errors via interfaces or tools like Hardhat/Foundry.

Proof of Concept:

```
function mulWadUp(uint256 x, uint256 y) internal pure returns (uint256
    z) {
    assembly {
        if mul(y, gt(x, div(not(0), y))) {
            // Incorrect error selector
            mstore(0x40, 0xbac65e5b)
            revert(0x1c, 0x04)
        }
        z := div(mul(x, y), WAD)
    }
}
```

Comparing this to the correct selector:

```
cast sig "MathMasters__mulWadFailed()"
# => 0xa56044f7
```

Recommended Mitigation: use the correct function selector:

```
-mstore(0x40, 0xbac65e5b)
+mstore(0x40, 0xa56044f7)
```

[L-04] Incorrect Constant in `sqrt` Initial Estimate Calculation (Potential Edge Case Inaccuracy)

Description:
The `sqrt` function uses a sequence of constant thresholds to generate a good initial guess for the Babylonian method. These constants are meant to be "all F" masks in hexadecimal form to ensure predictable bit-shifting behavior. However, one of the constants — `16777002` — is incorrect:

```
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))))
        // Correct: 16777215 0xffffff
@>      r := or(r, shl(4, lt(16777002, shr(r, x))))
        z := shl(shr(1, r), z)
        ...
    }
}
```

Here's the issue: - `87112285931760246646623899502532662132735` = `0xFFFFFFFFFFFFFFFFFFFFFFF` - `4722366482869645213695` = `0xFFFFFFFFFFFFFFFFF` - `1099511627775` = `0xFFFFFFFFFF` - `16777002` = `0xFFFF2A` (**incorrect** — should be `0xFFFFFF` / `16777215`)

While the function still produces correct results for most inputs, the use of `0xFFFF2A` instead of `0xFFFFFF` introduces a subtle mismatch that could affect the branch condition for extremely specific values of `x`, potentially altering the initial guess and, in very rare cases, impacting the number of iterations or the final output.

Impact:

- Very low probability of incorrect square root for edge case values of `x` that lie near the incorrect threshold. - Possible deviation in performance characteristics (more iterations needed) for those edge cases. - Reduced maintainability and readability since the constant breaks the expected "all F" pattern.

Recommended Mitigation:

Replace the incorrect constant `16777002` with the correct `16777215` (`0xFFFFFF`):

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

This ensures all constants follow the intended "full mask" pattern and avoids potential precision or iteration count issues in rare cases.