MathMasters Security Audit Report by MK (14th DEC 2024)

High Risk Findings

[H-1] Incorrect Rounding Logic in `mulWadUp` Function

Description:

The function `MathMasters::mulWadUp` accepts two input parameters (`uint256 x` and `uint256 y`) and calculates the expression `x * y / 1e18` and rounds the result up. There is a `if` statement in the function that increments the `x` value with `1`. Maybe, the reason for doing that is the result to be rounded up, but this is incorrect.

Impact:

This error can cause unexpected behavior in contracts relying on precise arithmetic, potentially leading to incorrect calculations, loss of funds, or unexpected outcomes in financial applications.

Proof of Concept:

1. By altering the settings of fuzz runs in `foundry.toml`in the following fashion:

```
[fuzz]
- runs = 100
+ runs = 10000
# runs = 1000000 # Use this one for prod
max_test_rejects = 65536
seed = '0x1'
dictionary_weight = 40
include_storage = true
include_push_bytes = true
extra_output = ["storageLayout", "metadata"]
```

2. And running `forge test --mt testMulWadUpFuzz` we find a test case where the expected result is different from what the function returns.

Recommended Mitigation:

To determine whether to round the result of the division up, we simply need to check if the following statement is true: `x * y % WAD != 0`

In the case of it being true then we must add `1` to the result of `x * y / WAD`. This is already done in the function, therefore we must simply remove the line above it which performs an unnecessary operation.

```
 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, or(div(not(0), y), x))) {
                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))
        }
    }
```

[H-2] Incorrect Overflow Check in `mulWadUp` Function

Description: In the `MathMasters::mulWadUp` function, the overflow check for the multiplication of `x * y` is ineffective, allowing the operation to proceed without reverting when an overflow occurs.

Impact: The ineffective overflow check in the mulWadUp function can result in incorrect calculations, as the multiplication may silently overflow. This poses a significant risk in applications that rely on accurate arithmetic, particularly in financial contexts, where such errors could lead to misallocated funds, loss of value, or exploitable vulnerabilities.

Proof of Concept:

1. By inserting the following test in `MathMasters.t.sol`

```
    function testMulWadUpOverFlowCheckFuzz(uint256 x, uint256 y) public {
        unchecked {
            if (x != 0 && (x * y) / x != y) {
                vm.expectRevert();
                MathMasters.mulWadUp(x, y);
            }
        }
    }
```

2. And running `forge test --mt testMulWadUpOverFlowCheckFuzz` we get that the function will return an invalid result when the expected behavior would be to revert.

Recommended Mitigation:

To resolve this issue, it is recommended that the overflow check for `x * y` in `MathMasters::mulWadUp` be updated to match the one implemented in `MathMasters::mulWad`.

```
    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, or(div(not(0), y), x))) {
+            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 Risk Findings

[L-1] Contract incompatible with solidity 0.8.3

Description: The codebase is intended to be used with any solidity version `^0.8.3`, however, the codebase uses custom errors that were only introduced in a later version. This project uses custom errors which is a feature that was introduced in solidity version `0.8.4`. Therefore an attempt to compile `MathMasters.sol` with solidity version `0.8.3` will result in a compilation error.

Impact: This mismatch results in a compilation error, preventing the project from being built on earlier versions of Solidity. Users must upgrade to version 0.8.4 or higher to resolve this issue and ensure proper compilation.

Proof of Concept:

1. If we attempt to compile a smart contract that uses version `0.8.3` of Solidity and includes the library, a compilation error will occur.
2. Let's create a new smart contract in `src/MathMastersExposed.sol` with a version`0.8.3` of Solidity that will include the library:

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity 0.8.3;

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

contract MathMastersExposed {
    using MathMasters for uint256;
}
```

3. If we attempt to compile this smart contract using `forge build`, we will encounter a compilation error:

```
Compiler run failed:
Error (2314): Expected ';' but got '('
 --> src/CustomErrors.sol:5:36:
  |
5 |     error MathMasters__MulWadFailed();
```

```
   |                                        ^

Error (2314): Expected ';' but got '('
  --> src/MathMasters.sol:14:41:
   |
14 |     error MathMasters__FactorialOverflow();
   |
```

Recommended Mitigation:

1. The library's pragma should not include version `0.8.3`.
2. Recommended changes to the `MathMasters.sol` library:

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

3. If we change the pragma from `0.8.3` to `0.8.4` in our previously created smart contract, we can now compile it successfully.

[L-2] Incorrect Memory Offset Causing Empty Return Data in `mulWad` and `mulWadUp`

Description: Both `mulWad` and `mulWadUp` functions incorrectly store the error selector hash at memory location `0x40` but use an offset of `0x1c` in the `revert()` call. This results in the revert returning empty data instead of the expected custom error selector hash. Correcting the memory location and offset ensures the proper error data is returned, aiding in debugging and improving error handling.

Impact: The incorrect memory offset causes the revert to return empty data instead of the expected custom error selector hash. This makes it difficult to identify the specific error that occurred, hindering debugging efforts. As a result, developers may struggle to diagnose and resolve issues effectively, impacting the reliability and transparency of the contract.

Recommended Mitigation: Store the error selector hash at the correct memory location, such as `0x80`, and use the proper offset, like `0x9c`, with a size of `0x04` when reverting. This ensures the custom error selector hash is returned correctly during a revert.