SPEARBIT

Primitive Security Review

Auditors

Kurt Barry, Lead Security Researcher

Christoph Michel, Lead Security Researcher

Report prepared by: Pablo Misirov

August 28, 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 5 days in total, Primitive engaged with Spearbit to review the portfolio protocol. In this period of time a total of **13** issues were found.

Summary

Project Name	Primitive
Repository	portfolio
Commit	f8302e...21de
Type of Project	Portfolio Management, DeFi
Audit Timeline	July 24 - July 28
Two week fix period	July 28 - Aug 7

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	1	1	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	3	2	1
Gas Optimizations	2	2	0
Informational	6	5	1
Total	**13**	**11**	**2**

5 Findings

5.1 Critical

5.1.1 tradingFunction returns wrong invariant at bounds, allowing to steal all pool reserves

Severity: *Critical Risk*

Context: NormalStrategyLib.sol#L157-L165

Description: The tradingFunction computing the invariant value of k = Φ¹(y/K) - Φ¹(1-x) + στ returns the wrong value at the bounds of x and y. The bounds of x are 0 and 1e18, the bounds of y are 0 and K, the strike price. If x or y is at these bounds, the corresponding term's computation is skipped and therefore implicitly set to 0, its initialization value.

```
int256 invariantTermX; // Φ¹(1-x)
// @audit if x is at the bounds, the term remains 0
if (self.reserveXPerWad.isBetween(lowerBoundX + 1, upperBoundX - 1)) {
  invariantTermX = Gaussian.ppf(int256(WAD - self.reserveXPerWad));
}
int256 invariantTermY; // Φ¹(y/K)
// @audit if y is at the bounds, the term remains 0
if (self.reserveYPerWad.isBetween(lowerBoundY + 1, upperBoundY - 1)) {
  invariantTermY = Gaussian.ppf(
    int256(self.reserveYPerWad.divWadUp(self.strikePriceWad))
  );
}
```

Note that Φ¹ = Gaussian.ppf is the probit function which is undefined at 0 and 1.0, but tends towards -infinity at 0 and +infinity at 1.0 = 1e18. (The closest values used in the Solidity approximation are Gaussian.ppf(1) = -8710427241990476442 ~ -8.71 and Gaussian.ppf(1e18-1) = 8710427241990476442 ~ 8.71.)

This fact can be abused by an attacker to steal the pool reserves. For example, the y-term Φ¹(y/K) will be a **negative value** for y/K < 0.5. Trading out all y reserve, will compute the new invariant with y set to 0 and the y-term Φ¹(y/K) = Φ¹(0) = -infinity is set to 0 instead, increasing the overall invariant, accepting the swap.

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

import "solmate/utils/SafeCastLib.sol";
import "./Setup.sol";

contract TestSpearbit is Setup {
  using SafeCastLib for uint256;
  using AssemblyLib for uint256;
  using AssemblyLib for uint128;
  using FixedPointMathLib for uint256;
  using FixedPointMathLib for uint128;

  function test_swap_all_out()
    public
    defaultConfig
    useActor
    usePairTokens(10 ether)
    allocateSome(1 ether)
  {
    (uint256 reserveAsset, uint256 reserveQuote) =
      subject().getPoolReserves(ghost().poolId);

      bool sellAsset = true;
      uint128 amtIn = 2; // pass reserve-not-stale check after taking fee
      uint128 amtOut = uint128(reserveQuote);
```

```
   uint256 prev = ghost().quote().to_token().balanceOf(actor());
   Order memory order = Order({
    useMax: false,
    poolId: ghost().poolId,
    input: amtIn,
    output: amtOut,
    sellAsset: sellAsset
   });
   subject().swap(order);
   uint256 post = ghost().quote().to_token().balanceOf(actor());
   assertTrue(post > prev, "swap-failed");
 }

}
```

Recommendation: The terms for values at the bounds $\Phi^{1}(y/K)$ and $\Phi^{1}(1-x)$ may not be set to zero as they would mathematically correspond to `+/- infinity`, resulting in a wrong invariant value. Swapping out all reserves should not be possible. As there are several other problems with one reserve value being zero, we recommend disallowing swapping out all reserves.

> Note that deallocating should already not allow zeroing a reserve as some initial LP tokens are locked and the deallocated amounts are rounded down.

Primitive: Fixed commit 743829.

- Checks if either reserve is gte a respective bound. If it is, set it as close to the bound as possible, but not at the bound so the Gaussian.ppf function does not revert.
- If one of the reserves is set very close to its bound, the other reserve will need to be very close to its opposite bound, else the invariant will be very negative.
- Adds a check in adjustReserves which gets triggered during a swap to revert if either virtualX or virtualY are zero.
- Removes the overwritten deltaLiquidity value that would allow 0 allocates to happen in getPoolMaxLiquidity.
- Also note that I updated the 'min delta' value in the invariant tests. It looks like if either of the reserves are changed by >= 3 wei then the trading function is strictly monotonic. If the delta is 2 or less, there's some cases where the invariant does not change.

Spearbit: Fixed.

5.2 Medium Risk

5.2.1 `getSpotPrice`, `approximateReservesGivenPrice`, `getStrategyData` ignore time to maturity

Severity: *Medium Risk*

Context: NormalStrategyLib.sol#L375

Description: When calling `getSpotPrice`, `getStrategyData` or `approximateReservesGivenPrice`, the pool config is `transform`ed into a `NormalCurve` struct. This transformation always sets the time to maturity field to the entire duration

```
function transform(PortfolioConfig memory config)
  pure
  returns (NormalCurve memory)
{
  return NormalCurve({
    reserveXPerWad: 0,
    reserveYPerWad: 0,
    strikePriceWad: config.strikePriceWad,
    standardDeviationWad: config.volatilityBasisPoints.bpsToPercentWad(),
    timeRemainingSeconds: config.durationSeconds,
    invariant: 0
  });
}
```

Neither is the `curve.timeRemainingSeconds` value overridden with the correct value for the mentioned functions. The reported spot price will be wrong after the pool has been initialized and integrators cannot rely on this value.

Recommendation: Initialize the `timeRemainingSeconds` value in `transform` to the current time remainings value or set it to the correct value afterwards for functions where it is needed. It should use a value similar to what `computeTau(..., block.timestamp)` returns. Consider adding additional tests for the affected functions for pools that have been active for a while.

Primitive: Fixed in commit 15ee0f.

Spearbit: Fixed. It was changed for `getSpotPrice`, comments have been added to the other functions.

5.3 Low Risk

5.3.1 Numerical error on larger trades favors the swapper relative to mathematically ideal pricing

Severity: *Low Risk*

Context: File.sol#L123

Description: To test the accuracy of the Solidity numerical methods used, a Python implementation of the swap logic was created using a library that supports arbitrary precision (`https://mpmath.org/`). Solidity swap executions generated in a custom fuzz test were compared against arbitrary precision results using Foundry's `ffi` feature (`https://book.getfoundry.sh/forge/differential-ffi-testing`). Cases where the "realized" swap price was better for the swapper than the "ideal" swap price were flagged. Deviations in the swapper's favor as large as 25% were observed (and larger ones likely exist). These seem to be a function of the size of the swap made--larger swaps favor the swapper more than smaller swaps (in fact, deviations were observed to trend towards zero as swap size relative to pool size decreased). It is unclear if there's any problem in practice from this behavior--large swaps will still incur large slippage and are only incentivized when the price has "jumped" drastically; fees also help make up for losses. Without going further, it can be stated that there is a risk for pools with frequent discontinuous price changes to track the theoretical payoff more poorly, but further numerical investigations are needed to determine whether there's a serious concern.

The test cases below require the simulation repo to be cloned into a Python virtual environment in a directory named `primitive-math-venv` with the needed dependencies at the same directory hierarchy level as the `portfolio` repository. That is, the `portfolio/` directory and `primitive-math-venv/` directories should be in the same folder, and the `primitive-math-venv/` folder should contain the `primitive-sim` repository. The virtual environment needs to be activated and have the `mpmath`, `scipy`, `numpy`, and `eth_abi` dependencies installed via `pip` or another method. Alternatively, these can be installed globally in which case the `primitive-math-venv` directory does not need to be a virtual environment.

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

import "solmate/utils/SafeCastLib.sol";
import "./Setup.sol";
```

```
contract TestNumericalDeviation is Setup {
  using SafeCastLib for uint256;
  using AssemblyLib for uint256;
  using AssemblyLib for uint128;
  using FixedPointMathLib for uint256;
  using FixedPointMathLib for uint128;

  bool printLogs = true;

  function _fuzz_random_args(
    bool sellAsset,
    uint256 amountIn,
    uint256 amountOut
  ) internal returns (bool swapExecuted) {
      Order memory maxOrder =
        subject().getMaxOrder(ghost().poolId, sellAsset, actor());

      amountIn =
        bound(amountIn, maxOrder.input / 1000 + 1, maxOrder.input);

      amountOut =
        subject().getAmountOut(ghost().poolId, sellAsset, amountIn, actor());
      if (printLogs) console.log("amountOut: ", amountOut);

      Order memory order = Order({
        useMax: false,
        poolId: ghost().poolId,
        input: amountIn.safeCastTo128(),
        output: amountOut.safeCastTo128(),
        sellAsset: sellAsset
      });

      try subject().simulateSwap({
        order: order,
        timestamp: block.timestamp,
        swapper: actor()
      }) returns (bool swapSuccess, int256 prev, int256 post) {
          try subject().swap(order) {
              assertTrue(
                  swapSuccess, "simulateSwap-failed but swap succeeded"
              );
              assertTrue(post >= prev, "post-invariant-not-gte-prev");
              swapExecuted = true;
          } catch {
              assertTrue(
                  !swapSuccess, "simulateSwap-succeeded but swap failed"
              );
          }
      } catch {
          // pass this case
      }
  }

    struct TestVals {
      uint256 strike;
      uint256 volatility_bps;
      uint256 durationSeconds;
      uint256 ttm;
    }

    // fuzzing entrypoint used to find violating swaps
    function test_swap_deviation(uint256 amtIn, uint256 amtOut)
```

```
        public
        defaultConfig
        useActor
        usePairTokens(10 ether)
        allocateSome(1 ether)
    {
        PortfolioPool memory pool = ghost().pool();
        (uint256 preXPerL, uint256 preYPerL) = (pool.virtualX, pool.virtualY);
        if (printLogs) {
            console.log("x_start: ", preXPerL);
            console.log("y_start: ", preYPerL);
        }

        TestVals memory tv;
        {
        uint256 creationTimestamp;
        (tv.strike, tv.volatility_bps, tv.durationSeconds, creationTimestamp,)
            = NormalStrategy(pool.strategy).configs(ghost().poolId);
        tv.ttm = creationTimestamp + tv.durationSeconds - block.timestamp;
        if (printLogs) {
            console.log("strike: ", tv.strike);
            console.log("volatility_bps: ", tv.volatility_bps);
            console.log("durationSeconds: ", tv.durationSeconds);
            console.log("creationTimestamp: ", creationTimestamp);
            console.log("block.timestamp: ", block.timestamp);
            console.log("ttm: ", tv.ttm);
            console.log("protocol fee: ", subject().protocolFee());
            console.log("pool fee: ", pool.feeBasisPoints);
            console.log("pool priority fee: ", pool.priorityFeeBasisPoints);
        }
        }

        bool sellAsset = true;
        if (printLogs) console.log("sellAsset: ", sellAsset);

        {
        bool swapExecuted = _fuzz_random_args(sellAsset, amtIn, amtOut);
        if (!swapExecuted) return;  // not interesting to check swap if it didn't execute
        }

        pool = ghost().pool();
        (uint256 postXPerL, uint256 postYPerL) = (pool.virtualX, pool.virtualY);
        if (printLogs) {
            console.log("x_end: ", postXPerL);
            console.log("y_end: ", postYPerL);
        }

        string[] memory cmds = new string[](18);
        cmds[0]  = "python3";
        cmds[1]  = "../primitive-math-venv/primitive-sim/check_swap_result.py";
        cmds[2]  = "--x";
        cmds[3]  = vm.toString(preXPerL);
        cmds[4]  = "--y";
        cmds[5]  = vm.toString(preYPerL);
        cmds[6]  = "--strike";
        cmds[7]  = vm.toString(tv.strike);
        cmds[8]  = "--vol_bps";
        cmds[9]  = vm.toString(tv.volatility_bps);
        cmds[10] = "--duration";
        cmds[11] = vm.toString(tv.durationSeconds);
        cmds[12] = "--ttm";
        cmds[13] = vm.toString(tv.ttm);
        cmds[14] = "--xprime";
        cmds[15] = vm.toString(postXPerL);
        cmds[16] = "--yprime";
        cmds[17] = vm.toString(postYPerL);
        bytes memory result = vm.ffi(cmds);
        (uint256 idealFinalDependentPerL) = abi.decode(result, (uint256));
        if (printLogs) console.log("idealFinalDependentPerL: ", idealFinalDependentPerL);

        uint256 postDependentPerL = sellAsset ? postYPerL : postXPerL;

        // Only worried if swap was _better_ than ideal
        if (idealFinalDependentPerL > postDependentPerL) {
            uint256 diff = idealFinalDependentPerL - postDependentPerL;
            uint256 percentErrWad = diff * 1e18 / idealFinalDependentPerL;

            if (printLogs) console.log("%% err wad: ", percentErrWad);
            // assert at worst 25% error
            assertLt(percentErrWad, 0.25 * 1e18);
        }
    }

    function test_swap_gt_2pct_dev_in_swapper_favor()
        public
        defaultConfig
        useActor
        usePairTokens(10 ether)
        allocateSome(1 ether)
    {
        uint256 amtIn  = 6552423086988641261559668799172253742131420409793952225706522955;
        uint256 amtOut = 0;

        PortfolioPool memory pool = ghost().pool();
        (uint256 preXPerL, uint256 preYPerL) = (pool.virtualX, pool.virtualY);
        if (printLogs) {
            console.log("x_start: ", preXPerL);
            console.log("y_start: ", preYPerL);
        }

        TestVals memory tv;
        {
        uint256 creationTimestamp;
        (tv.strike, tv.volatility_bps, tv.durationSeconds, creationTimestamp,)
            = NormalStrategy(pool.strategy).configs(ghost().poolId);
        tv.ttm = creationTimestamp + tv.durationSeconds - block.timestamp;
        if (printLogs) {
            console.log("strike: ", tv.strike);
            console.log("volatility_bps: ", tv.volatility_bps);
            console.log("durationSeconds: ", tv.durationSeconds);
            console.log("creationTimestamp: ", creationTimestamp);
            console.log("block.timestamp: ", block.timestamp);
            console.log("ttm: ", tv.ttm);
            console.log("protocol fee: ", subject().protocolFee());
            console.log("pool fee: ", pool.feeBasisPoints);
            console.log("pool priority fee: ", pool.priorityFeeBasisPoints);
        }
        }

        bool sellAsset = true;
        if (printLogs) console.log("sellAsset: ", sellAsset);

        {
        bool swapExecuted = _fuzz_random_args(sellAsset, amtIn, amtOut);
        if (!swapExecuted) return;  // not interesting to check swap if it didn't execute
        }

        pool = ghost().pool();
        (uint256 postXPerL, uint256 postYPerL) = (pool.virtualX, pool.virtualY);
        if (printLogs) {
            console.log("x_end: ", postXPerL);
            console.log("y_end: ", postYPerL);
        }

        string[] memory cmds = new string[](18);
        cmds[0]  = "python3";
        cmds[1]  = "../primitive-math-venv/primitive-sim/check_swap_result.py";
        cmds[2]  = "--x";
        cmds[3]  = vm.toString(preXPerL);
        cmds[4]  = "--y";
        cmds[5]  = vm.toString(preYPerL);
        cmds[6]  = "--strike";
        cmds[7]  = vm.toString(tv.strike);
        cmds[8]  = "--vol_bps";
        cmds[9]  = vm.toString(tv.volatility_bps);
        cmds[10] = "--duration";
        cmds[11] = vm.toString(tv.durationSeconds);
        cmds[12] = "--ttm";
        cmds[13] = vm.toString(tv.ttm);
        cmds[14] = "--xprime";
        cmds[15] = vm.toString(postXPerL);
        cmds[16] = "--yprime";
        cmds[17] = vm.toString(postYPerL);
        bytes memory result = vm.ffi(cmds);
        (uint256 idealFinalYPerL) = abi.decode(result, (uint256));
        if (printLogs) console.log("idealFinalYPerL: ", idealFinalYPerL);

        // Only worried if swap was _better_ than ideal
        if (idealFinalYPerL > postYPerL) {
            uint256 diff = idealFinalYPerL - postYPerL;
            uint256 percentErrWad = diff * 1e18 / idealFinalYPerL;

            if (printLogs) console.log("%% err wad: ", percentErrWad);
            // assert at worst 2% error
            assertLt(percentErrWad, 0.02 * 1e18);
        }
    }

    function test_swap_gt_5pct_dev_in_swapper_favor()
        public
        defaultConfig
        useActor
        usePairTokens(10 ether)
        allocateSome(1 ether)
    {
        uint256 amtIn  = 524204019310836059902749478707356665714276202503631350973429403;
        uint256 amtOut = 0;

        PortfolioPool memory pool = ghost().pool();
        (uint256 preXPerL, uint256 preYPerL) = (pool.virtualX, pool.virtualY);
        if (printLogs) {
            console.log("x_start: ", preXPerL);
            console.log("y_start: ", preYPerL);
        }

        TestVals memory tv;
```

```
{
uint256 creationTimestamp;
(tv.strike, tv.volatility_bps, tv.durationSeconds, creationTimestamp,)
    = NormalStrategy(pool.strategy).configs(ghost().poolId);
tv.ttm = creationTimestamp + tv.durationSeconds - block.timestamp;
if (printLogs) {
    console.log("strike: ", tv.strike);
    console.log("volatility_bps: ", tv.volatility_bps);
    console.log("durationSeconds: ", tv.durationSeconds);
    console.log("creationTimestamp: ", creationTimestamp);
    console.log("block.timestamp: ", block.timestamp);
    console.log("ttm: ", tv.ttm);
    console.log("protocol fee: ", subject().protocolFee());
    console.log("pool fee: ", pool.feeBasisPoints);
    console.log("pool priority fee: ", pool.priorityFeeBasisPoints);
}
}

bool sellAsset = true;
if (printLogs) console.log("sellAsset: ", sellAsset);

{
bool swapExecuted = _fuzz_random_args(sellAsset, amtIn, amtOut);
if (!swapExecuted) return;  // not interesting to check swap if it didn't execute
}

pool = ghost().pool();
(uint256 postXPerL, uint256 postYPerL) = (pool.virtualX, pool.virtualY);
if (printLogs) {
    console.log("x_end: ", postXPerL);
    console.log("y_end: ", postYPerL);
}

string[] memory cmds = new string[](18);
cmds[0]  = "python3";
cmds[1]  = "../primitive-math-venv/primitive-sim/check_swap_result.py";
cmds[2]  = "--x";
cmds[3]  = vm.toString(preXPerL);
cmds[4]  = "--y";
cmds[5]  = vm.toString(preYPerL);
cmds[6]  = "--strike";
cmds[7]  = vm.toString(tv.strike);
cmds[8]  = "--vol_bps";
cmds[9]  = vm.toString(tv.volatility_bps);
cmds[10] = "--duration";
cmds[11] = vm.toString(tv.durationSeconds);
cmds[12] = "--ttm";
cmds[13] = vm.toString(tv.ttm);
cmds[14] = "--xprime";
cmds[15] = vm.toString(postXPerL);
cmds[16] = "--yprime";
cmds[17] = vm.toString(postYPerL);
bytes memory result = vm.ffi(cmds);
(uint256 idealFinalYPerL) = abi.decode(result, (uint256));
if (printLogs) console.log("idealFinalYPerL: ", idealFinalYPerL);

// Only worried if swap was _better_ than ideal
if (idealFinalYPerL > postYPerL) {
    uint256 diff = idealFinalYPerL - postYPerL;
    uint256 percentErrWad = diff * 1e18 / idealFinalYPerL;

    if (printLogs) console.log("%% err wad: ", percentErrWad);
```

```
            // assert at worst 2% error
            assertLt(percentErrWad, 0.05 * 1e18);
        }
    }

    function test_swap_gt_25pct_dev_in_swapper_favor()
        public
        defaultConfig
        useActor
        usePairTokens(10 ether)
        allocateSome(1 ether)
    {
        uint256 amtIn  = 110109023928019935126448015360767432374367360662791991077231763772041488708545;
        uint256 amtOut = 0;

        PortfolioPool memory pool = ghost().pool();
        (uint256 preXPerL, uint256 preYPerL) = (pool.virtualX, pool.virtualY);
        if (printLogs) {
            console.log("x_start: ", preXPerL);
            console.log("y_start: ", preYPerL);
        }

        TestVals memory tv;
        {
        uint256 creationTimestamp;
        (tv.strike, tv.volatility_bps, tv.durationSeconds, creationTimestamp,)
            = NormalStrategy(pool.strategy).configs(ghost().poolId);
        tv.ttm = creationTimestamp + tv.durationSeconds - block.timestamp;
        if (printLogs) {
            console.log("strike: ", tv.strike);
            console.log("volatility_bps: ", tv.volatility_bps);
            console.log("durationSeconds: ", tv.durationSeconds);
            console.log("creationTimestamp: ", creationTimestamp);
            console.log("block.timestamp: ", block.timestamp);
            console.log("ttm: ", tv.ttm);
            console.log("protocol fee: ", subject().protocolFee());
            console.log("pool fee: ", pool.feeBasisPoints);
            console.log("pool priority fee: ", pool.priorityFeeBasisPoints);
        }
        }

        bool sellAsset = true;
        if (printLogs) console.log("sellAsset: ", sellAsset);

        {
        bool swapExecuted = _fuzz_random_args(sellAsset, amtIn, amtOut);
        if (!swapExecuted) return;  // not interesting to check swap if it didn't execute
        }

        pool = ghost().pool();
        (uint256 postXPerL, uint256 postYPerL) = (pool.virtualX, pool.virtualY);
        if (printLogs) {
            console.log("x_end: ", postXPerL);
            console.log("y_end: ", postYPerL);
        }

        string[] memory cmds = new string[](18);
        cmds[0]  = "python3";
        cmds[1]  = "../primitive-math-venv/primitive-sim/check_swap_result.py";
        cmds[2]  = "--x";
        cmds[3]  = vm.toString(preXPerL);
        cmds[4]  = "--y";
        cmds[5]  = vm.toString(preYPerL);
        cmds[6]  = "--strike";
        cmds[7]  = vm.toString(tv.strike);
        cmds[8]  = "--vol_bps";
        cmds[9]  = vm.toString(tv.volatility_bps);
        cmds[10] = "--duration";
        cmds[11] = vm.toString(tv.durationSeconds);
        cmds[12] = "--ttm";
        cmds[13] = vm.toString(tv.ttm);
        cmds[14] = "--xprime";
        cmds[15] = vm.toString(postXPerL);
        cmds[16] = "--yprime";
        cmds[17] = vm.toString(postYPerL);
        bytes memory result = vm.ffi(cmds);
        (uint256 idealFinalYPerL) = abi.decode(result, (uint256));
        if (printLogs) console.log("idealFinalYPerL: ", idealFinalYPerL);

        // Only worried if swap was _better_ than ideal
        if (idealFinalYPerL > postYPerL) {
            uint256 diff = idealFinalYPerL - postYPerL;
            uint256 percentErrWad = diff * 1e18 / idealFinalYPerL;

            if (printLogs) console.log("%% err wad: ", percentErrWad);
            // assert at worst 25% error
            assertLt(percentErrWad, 0.25 * 1e18);
        }
    }
}
```

Recommendation: Do more thorough numerical testing to determine under what conditions swap pricing deviates from "ideal" and whether this is a practical concern.

Spearbit: No comment. Marking as acknowledged.

5.3.2 `getMaxOrder` overestimates output values

Severity: *Low Risk*

Context: NormalStrategy.sol#L230-L237

Description: The `getMaxOrder` function adds `+ 1` to the output value, overestimating the output value. This can lead to failed swaps if this value is used.

```
tempOutput = pool.virtualY - lowerY.mulWadDown(pool.liquidity) + 1;
```

It's also easy to see that with `lowerY = 0` we have `tempOutput = pool.virtualY - lowerY.mulWadDown(pool.liquidity) + 1 = pool.virtualY + 1`, i.e., the max out amount would be *more* than the pool reserves.

Recommendation: Consider subtracting 1 instead of adding 1.

Primitive: Fixed in commit f0b6d4.

Spearbit: Fixed.

5.3.3 Improve reentrancy guards

Severity: *Low Risk*

Context: Portfolio.sol#L124

Description: Previously, only `settlement` performed calls to arbitrary addresses through ERC20 transfers. With recent additions, like the `ERC1155._mint` and user-provided strategies, single actions like `allocate` and `swap` also perform calls to potentially malicious contracts. This increases the attack surface for reentrancy attacks.

The current way of protecting against reentrancy works by setting multicall flags (`_currentMulticall`) and locks (`preLock()` and `postLock()`) on multicalls and single-action calls. However, the single calls essentially skip reentrancy guards if the outer context is a multicall. This still allows for reentrancy through control flows like the following:

```
// reenter during multicall's action execution
multicall
  preLock()
    singleCall()
      reenter during current execution
        singeCall()
          preLock(): passes because we're in multicall
          skips settlement
          postLock(): passes because we're in multicall
  _currentMulticall = false;
  settlement()
  postLock()

// reenter during multicall's settlement
multicall
  preLock()
    singleCall
      preLock(): ...
      postLock(): `_locked = 1`
  _currentMulticall = false;
  settlement()
    reenter
      singeCall()
        passes preLock because not locked
      mutliCall()
        passes multicall reentrancy guard because not in multicall
        passes preLock because not locked
  ... settlement finishes
  postLock()
```

Recommendation: While it's not obvious how to exploit the reentrancy issues, we propose improving the reentrancy guards. The main issue is that the current reentrancy lock has no "depth information", it does not know if an action that is run during a multicall execution is part of the original scheduled multicall actions or an injected one. By adjusting the `_locked` field to count the reentrancy call depth (instead of resetting it on *any* `_preLock`/`_postLock`) we can ensure that only the originally defined multicall actions are run.

```
 function _preLock() private {
-   if (_locked != 1 && !_currentMulticall) {
-       revert Portfolio_InvalidReentrancy();
-   }
+    // if it has been locked before: `_locked != 1`
+    // revert if not in a multicall. `!_currentMulticall` (singleCall -> singleCall reentrancy)
+    // or revert if in a multicall and this was called from another singleCall. `_locked > 2`
↪   (multiCall -> singleCall -> singleCall reentrancy)
+    if (_locked != 1 && (!_currentMulticall || _locked > 2)) {
+        revert Portfolio_InvalidReentrancy();
+    }

-   _locked = 2;
+   _locked++;
 }

 function _postLock() private {
-   _locked = 1;
+   _locked--;

    // Reverts if the account system was not settled after a normal call.
    if (!__account__.settled && !_currentMulticall) {
        revert Portfolio_InvalidSettlement();
    }
 }
```

Note that after a successful transaction, the `_locked` field is reset to its original value as each `_preLock` has a single corresponding `_postLock`, and vice versa.

Primitive: Fixed in commit 4bf82d.

Spearbit: Fixed. Recommendation was implemented.

5.4 Gas Optimization

5.4.1 `approximatePriceGivenX` does not need to compute y-bounds

Severity: *Gas Optimization*

Context: NormalStrategyLib.sol#L308

Description: The `approximatePriceGivenX` function does not need to compute the y-bounds by calling `self.getReserveYBounds()`.

Recommendation: Consider removing this call.

Primitive: Fixed in commit f994c2.

Spearbit: Fixed.

5.4.2 Unnecessary computations in `NormalStrategy.beforeSwap`

Severity: *Gas Optimization*

Context: NormalStrategy.sol#L87

Description: The `NormalStrategy.sol.beforeSwap` function calls `getSwapInvariants` to simulate an entire swap with current and post-swap invariants. However, only the current invariant value is used.

Recommendation: Consider calculating the current invariant only instead of calling `getSwapInvariants`. This also eliminates having to create a fake order with an `input` and `output` of 2 to make the swap simulation pass. The invariant computation should use the same rounding as done in `getSwapInvariants`.

Primitive: Fixed in commit a39126.

Spearbit: Fixed.

5.5 Informational

5.5.1 Pools can use malicious strategies

Severity: *Informational*

Context: Portfolio.sol#L671

Description: Anyone can create pools and configure the pool to use a custom strategy. A malicious strategy can disable swapping and (de-)allocating at any time, as well as enable privileged parties to trade out all pool reserves by implementing custom logic in the `validateSwap` function.

Recommendation: When users trade or provide liquidity in unofficial strategies they risk losing their funds. Users should thoroughly check the strategy of the pool before engaging with it.

Primitive: Fixed here commit acb0cf.

They were in the right place in createPool, but the function arguments for encode were in the wrong place so I switched them around. Also added tests for all the combinations.

Spearbit: Acknowledged.

5.5.2 `findRootForSwappingIn` functions should use MINIMUM_INVARIANT_DELTA

Severity: *Informational*

Context: NormalStrategyLib.sol#L654, NormalStrategy.sol#L11

Description: The `findRootForSwappingInX` and `findRootForSwappingInY` functions add `+ 1` to the previous curve invariant

```
tradingFunction(curve) - (curve.invariant + 1)
```

Recommendation: Instead of adding `+ 1`, consider using adding `+ MINIMUM_INVARIANT_DELTA` instead. Finding a root for this function then yields the desired output of finding curve values such that

```
newInvar - (prevInvar + delta) >= 0
newInvar - prevInvar - delta >= 0
newInvar - prevInvar >= delta
```

This matches the swap invariant check in `_validateSwap`.

Primitive: Fixed in commit 973230.

Spearbit: Fixed.

5.5.3 Unused Errors

Severity: *Informational*

Context: NormalStrategyLib.sol#L48-L49

Description: The `NormalStrategyLib_UpperPriceLimitReached` and `NormalStrategyLib_LowerPriceLimitReached` errors are not used.

Recommendation: Consider using these errors or removing them.

Primitive: Fixed in commit 3bd709.

Spearbit: Fixed.

5.5.4 `getSwapInvariants` order output can be `1` instead of `2`

Severity: *Informational*

Context: NormalStrategy.sol#L91, NormalStrategy.sol#L180, SwapLib.sol#L133

Description: The `getSwapInvariants` function is used to simulate swaps for the `getAmountOut` and `beforeSwap` functions. These functions use an artificial output value of `2` such that the function does not revert.

Recommendation: The `output` value can be reduced to `1` instead as there is no fee on the output value and the only check on the output reserves is that they changed.

Primitive: Commit fix commit 797457.

This was then adjusted to remove the unnecessary computations in `getSwapInvariant`: commit a39126.

Spearbit: Fixed in `beforeSwap` because the calls changed. fixed in `getAmountOut`.

5.5.5 `AfterCreate` event uses wrong `durationSeconds` value if pool is perpetual

Severity: *Informational*

Context: NormalStrategy.sol#L74, NormalStrategyLib.sol#L408

Description: The `AfterCreate` uses the cached `config.durationSeconds` value but the real value the config storage struct is initialized with will be `SECONDS_PER_YEAR` in the case of perpetual pools.

Recommendation: Consider reading the `durationSeconds` value from the config storage var `configs[poolId]` instead.

Primitive: Fixed in commit 8e645a.

Spearbit: Fixed.

5.5.6 Unnecessary fee reserves check

Severity: *Informational*

Context: SwapLib.sol#L119

Description: The fee amount is always taken on the input and the fee percentage is always less than 100%. Therefore, the fee is always less than the input. The following check should never fail

```
adjustedInputReserveWad += self.input;
// feeAmountUnit <= self.input <= adjustedInputReserveWad
if (feeAmountUnit > adjustedInputReserveWad) revert SwapLib_FeeTooHigh();
```

Recommendation: Consider removing the check and adding a comment instead explaining why the fee is always included in the adjusted reserve and the subtraction doesn't underflow.

Primitive: Fixed in commit 728b04.

Spearbit: Fixed.