Wells (Basin) Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Giovanni Di Siena

Hans

Assisting Auditors

Alex Roan

Patrick Collins

June 16, 2023

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

The Beanstalk Wells protocol is a framework for deploying constant function automated market maker liquidity pools which can be configured to use any number of tokens and any Well function. Wells are designed to be used by the Beanstalk protocol but can be used by any other protocol or project due to the implementation of 'Pumps' which are on-chain oracles used to determine the pricing of tokens.

Every file in src was in scope for this audit, except for `ConstantProduct.sol`.

5 Executive Summary

Over the course of 11 days, the Cyfrin team conducted an audit on the Wells (Basin) smart contracts provided by Beanstalk. In this period, a total of 29 issues were found.

Summary

Project Name	Wells (Basin)
Repository	Basin
Commit	e5441fc78f0f...
Audit Timeline	Mar 10th - Mar 24th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	0
High Risk	4
Medium Risk	2
Low Risk	3
Informational	18
Gas Optimizations	2
Total Issues	29

Summary of Findings

[H-1] Protocol's invariants can be broken	Resolved
[H-2] Each Well is responsible for ensuring that an `update` call cannot be made with a reserve of 0	Acknowledged
[H-3] `removeLiquidity` logic is not correct for generalized Well functions other than ConstantProduct	Resolved
[H-4] Read-only reentrancy	Resolved
[M-1] Should ensure uniqueness of the tokens of Wells	Resolved
[M-2] `LibLastReserveBytes::storeLastReserves` has no check for reserves being too large	Acknowledged
[L-1] TWAP is incorrect when only 1 update has occurred	Closed
[L-2] Lack of validation for `A` in `GeoEmaAndCumSmaPump::constructor`	Resolved
[L-3] Incorrect sload in LibBytes	Resolved
[I-01] Non-standard storage packing	Resolved
[I-02] EIP-1967 second pre-image best practice	Resolved
[I-03] Remove experimental ABIEncoderV2 pragma	Resolved
[I-04] Inconsistent use of decimal/hex notation in inline assembly	Acknowledged
[I-05] Unused imports and errors	Resolved
[I-06] Inconsistency in LibMath comments	Resolved
[I-07] FIXME and TODO comments	Resolved
[I-08] Use correct NatSpec tags	Acknowledged
[I-09] Poorly descriptive variable and function names in `GeoEmaAndCumSmaPump` are difficult to read	Resolved
[I-10] Remove TODO Check if bytes shift is necessary	Resolved
[I-11] Use underscore prefix for internal functions	Resolved
[I-12] Missing test coverage for a number of functions	Resolved
[I-13] Use `uint256` over `uint`	Resolved

[I-14] Use constant variables in place of inline magic numbers	Resolved
[I-15] Insufficient use of NatSpec and comments on complex code blocks	Resolved
[I-16] Precision loss on large values transformed between log2 scale and the normal scale	Closed
[I-17] Emit events prior to external interactions	Acknowledged
[I-18] Time Weighted Average Price oracles are susceptible to manipulation	Acknowledged
[G-1] Simplify modulo operations	Resolved
[G-2] Branchless optimization	Resolved

6 Findings

6.1 High Risk

6.1.1 Protocol's invariants can be broken

Description: The protocol intends to provide a generalized framework for constant-function AMM liquidity pools. We have identified some invariants that should hold at any given time. One of these invariants is `totalSupply() == calcLpTokenSupply(reserves)`, and we can interpret this as the pool's total LP supply should match the calculation of LP from the current `reserves` state values.

This invariant can be broken with valid transactions in the current implementation, leading to several problems. For example, valid liquidity removal might revert, as shown in the PoC test below.

Impact: The impact is HIGH because this discrepancy will lead to protocol insolvency (revert on valid transactions).

Proof of Concept:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.17;

import {IERC20} from "test/TestHelper.sol";
import {IWellFunction} from "src/functions/ConstantProduct2.sol";
import {LiquidityHelper} from "test/LiquidityHelper.sol";
import "forge-std/Test.sol";
import {MockToken} from "mocks/tokens/MockToken.sol";

contract GetRemoveLiquidityOneTokenOutArithmeticFail is LiquidityHelper {
    function setUp() public {
        setupWell(2);
    }

    address internal constant ADDRESS_0 = 0x0000000000000000000000000000000000001908;
    address internal constant ADDRESS_1 = 0xffffFFFfFFffffffffffffffFfFFFffffFFFfFFfE;
    address internal constant ADDRESS_2 = 0x0000000000000000000000000000000000000001;
    address internal constant ADDRESS_3 = 0x00000011EF7b76c418fC426B8707dC60c8404f4a;

    // Failing unit tests from invariant results
    function test_getRemoveLiquidityOneTokenOutArithmeticFail() public {
        showStatus();

        address msgSender = ADDRESS_0;
        _addLiquidity(msgSender, 77_470_052_844_788_801_811_950_156_551, 17_435);
        _removeLiquidityOneTokenOut(msgSender, 5267, 0);
        showStatus();

        msgSender = ADDRESS_1;
        _addLiquidity(msgSender, 79_228_162_514_264_337_593_543_950_335, 0);
        _removeLiquidity(msgSender, 2_025_932_259_663_320_959_193_637_370_794);
        showStatus();

        msgSender = ADDRESS_2;
        _addLiquidity(msgSender, 69_069_904_726_099_247_337_000_262_288, 3);
        showStatus();

        msgSender = ADDRESS_1;
        _transferLp(msgSender, ADDRESS_3, 1_690_276_116_468_540_706_301_324_542_928);
        emit log_named_uint("LP Balance (1)", well.balanceOf(ADDRESS_1));
        emit log_named_uint("LP Balance (3)", well.balanceOf(ADDRESS_3));
        showStatus();
```

```
        msgSender = ADDRESS_0;
        emit log_string("removeLiquidity");
        _removeLiquidity(msgSender, 122_797_404_990_851_137_316_041_024_188);
        showStatus();

        msgSender = ADDRESS_3;
        emit log_string("removeLiquidityOneToken");
        _removeLiquidityOneTokenOut(msgSender, 1_690_276_116_468_540_706_301_000_000_000, 1);
        emit log_named_uint("LP Balance (3)", well.balanceOf(ADDRESS_3));
        showStatus();
        // The next line fails with an under/overflow error
        //
        // CONTEXT: In the previous operation, ADDRESS_3 removes the vast majority of his LP position
        // for token[1]. At this point the token balances of the well are as follows:
        //  * token[0].balanceOf(well) = 1985088525924048657164518345 87
        //  * token[1].balanceOf(well) =          625986797429655048967
        // The next operation ADDRESS_3 calls is removeLiquidityOneTokenOut() for token[0] using his
        // remaining LP position. The amount of LP tokens he has left is 3. Line 526 reverts with
        ↪  underflow,
        // despite all operations being completely valid. How severe is this?
        _removeLiquidityOneTokenOut(msgSender, 324_542_928, 0);
        showStatus();
    }

    function test_audit() public {
        address msgSender = address(this);
        _removeLiquidityOneTokenOut(msgSender, 1e16, 0);
        _removeLiquidity(msgSender, well.balanceOf(msgSender) - 1);
        _removeLiquidity(msgSender, 1);
    }

    function test_totalSupplyInvariantSwapFail() public {
        address msgSender = 0x576024f76bd1640d7399a5B5F61530f997Ae06f2;
        changePrank(msgSender);
        IERC20[] memory mockTokens = well.tokens();
        uint tokenInIndex = 0;
        uint tokenOutIndex = 1;
        uint tokenInAmount = 52_900_000_000_000_000_000;
        MockToken(address(mockTokens[tokenInIndex])).mint(msgSender, tokenInAmount);
        mockTokens[tokenInIndex].approve(address(well), tokenInAmount);
        uint minAmountOut = well.getSwapOut(mockTokens[tokenInIndex], mockTokens[tokenOutIndex],
        ↪  tokenInAmount);
        well.swapFrom(
            mockTokens[tokenInIndex], mockTokens[tokenOutIndex], tokenInAmount, minAmountOut,
            ↪  msgSender, block.timestamp
        );
        // check the total supply
        uint functionCalc =
            IWellFunction(well.wellFunction().target).calcLpTokenSupply(well.getReserves(),
            ↪  well.wellFunction().data);
        // assertEq(well.totalSupply(), functionCalc);
        showStatus();
    }

    function _transferLp(address msgSender, address to, uint amount) private {
        changePrank(msgSender);
        well.transfer(to, amount);
    }

    function _removeLiquidityOneTokenOut(
        address msgSender,
        uint lpAmountIn,
```

```
        uint tokenIndex
    ) private returns (uint tokenAmountOut) {
        changePrank(msgSender);
        IERC20[] memory mockTokens = well.tokens();
        uint minTokenAmountOut = well.getRemoveLiquidityOneTokenOut(lpAmountIn, mockTokens[tokenIndex]);
        tokenAmountOut = well.removeLiquidityOneToken(
            lpAmountIn, mockTokens[tokenIndex], minTokenAmountOut, msgSender, block.timestamp
        );
    }

    function _removeLiquidity(address msgSender, uint lpAmountIn) private returns (uint[] memory
    ↪  tokenAmountsOut) {
        changePrank(msgSender);
        uint[] memory minTokenAmountsOut = well.getRemoveLiquidityOut(lpAmountIn);
        tokenAmountsOut = well.removeLiquidity(lpAmountIn, minTokenAmountsOut, msgSender,
        ↪  block.timestamp);
    }

    function _addLiquidity(
        address msgSender,
        uint token0Amount,
        uint token1Amount
    ) private returns (uint lpAmountOut) {
        changePrank(msgSender);
        uint[] memory tokenAmountsIn = _mintToSender(msgSender, token0Amount, token1Amount);
        uint minLpAmountOut = well.getAddLiquidityOut(tokenAmountsIn);

        lpAmountOut = well.addLiquidity(tokenAmountsIn, minLpAmountOut, msgSender, block.timestamp);
    }

    function _mintToSender(
        address msgSender,
        uint token0Amount,
        uint token1Amount
    ) private returns (uint[] memory tokenAmountsIn) {
        changePrank(msgSender);
        IERC20[] memory mockTokens = well.tokens();
        MockToken(address(mockTokens[0])).mint(msgSender, token0Amount);
        MockToken(address(mockTokens[1])).mint(msgSender, token1Amount);

        tokenAmountsIn = new uint[](2);
        tokenAmountsIn[0] = token0Amount;
        tokenAmountsIn[1] = token1Amount;

        mockTokens[0].approve(address(well), token0Amount);
        mockTokens[1].approve(address(well), token1Amount);
    }

    function showStatus() public {
        IERC20[] memory mockTokens = well.tokens();
        uint[] memory reserves = well.getReserves();

        uint calcedSupply =
        ↪  IWellFunction(well.wellFunction().target).calcLpTokenSupply(well.getReserves(),
        ↪  well.wellFunction().data);
        emit log_named_uint("Total  LP Supply", well.totalSupply());
        emit log_named_uint("Calced LP Supply", calcedSupply);
    }
}
```

The test results are shown below.

```
Running 1 test for test/invariant/RemoveOneLiquidity.t.sol:GetRemoveLiquidityOneTokenOutArithmeticFail
[FAIL. Reason: Arithmetic over/underflow] test_getRemoveLiquidityOneTokenOutArithmeticFail() (gas:
↪  1029158)
Logs:
  Total  LP Supply: 100000000000000000000000000
  Calced LP Supply: 100000000000000000000000000
  Total  LP Supply: 880170743917274287191919288925
  Calced LP Supply: 880170743917274287191919288909
  Total  LP Supply: 104919835556412835782205138311854
  Calced LP Supply: 104919835556412835782222604328 21
  Total  LP Supply: 129604463462892404776192018028 43
  Calced LP Supply: 129604463462892404776192018028 43
  LP Balance (1): 1
  LP Balance (3): 169027611646854070630132454292 8
  Total  LP Supply: 129604463462892404776192018028 43
  Calced LP Supply: 129604463462892404776192018028 43
  removeLiquidity
  Total  LP Supply: 128376489412983893403031607786 55
  Calced LP Supply: 128376489412983893403104294643 50
  removeLiquidityOneToken
  LP Balance (3): 324542928
  Total  LP Supply: 111473728248298486340021607786 55
  Calced LP Supply: 111473728248298486339986812149 26

Test result: FAILED. 0 passed; 1 failed; finished in 3.70ms

Failing tests:
Encountered 1 failing test in
↪  test/invariant/RemoveOneLiquidity.t.sol:GetRemoveLiquidityOneTokenOutArithmeticFail
[FAIL. Reason: Arithmetic over/underflow] test_getRemoveLiquidityOneTokenOutArithmeticFail() (gas:
↪  1029158)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Looking into this, the `ConstantProduct2`, we see three problems:

1. No rounding direction is specified in the functions `calcReserve` and `calcLpTokenSupply`. These Well functions are used in critical places where the Well's state (reserve, LP) will change. Mathematical discrepancies should be rounded in favor of the protocol; however, no rounding direction is specified in the current implementation, which allows unfavorable transactions.

   For example, in Well::getSwapOut and Well::_getRemoveLiquidityOneTokenOut, the calculation of the output token amount is rounded up (downside rounded value is subtracted) while it should be rounded down. These unfavorable transactions will compound, affecting the health of the protocol.

2. This invariant is not guaranteed after `Well::removeLiquidity`. The current implementation calculates the token output amount from the ratio of the `lpAmountIn` to the `totalSupply`, assuming that `calcLpTokenSupply([r0 * delta, r1 * delta]) = totalSupply() * delta` where `delta` denotes the ratio `1 - lpAmountIn/lpTokenSupply`. This is also mentioned separately in another issue, but we want to note again that ensuring the invariant holds after any transaction is essential. Handling the last token separately is recommended, similar to `Well::removeLiquidityOneToken`.

3. It is practically almost impossible to make the invariant `totalSupply() == calcLpTokenSupply(reserves)` hold strictly. For this example, we will focus on the `Well::removeLiquidityOneToken` function because it is the simplest and most reasonable way to withdraw liquidity, and we will assume `ConstantProduct2` is the Well function. The mechanism is intuitive - the protocol burns the requested `lpAmountIn` and calculates the new reserve value for the requested `tokenOut` (`tokens[0]`) using the decreased LP token supply.

Let us assume `totalSupply() == calcLpTokenSupply(reserves) == T0` before the transaction. After

the transaction, the total supply will be `T0 - lpAmountIn`. The output token amount is calculated as `getRemoveLiquidityOneTokenOut(lpAmountIn, 0, reserves) = reserves[0] - calcReserve(reserves, 0, T0 - lpAmountIn)`. After the transaction, the calculated total supply will be `calcLpTokenSupply([calcReserve(reserves, 0, T0 - lpAmountIn), reserves[1]])`. For the invariant to hold after the transaction, the functions `ConstantProduct2::calcLpTokenSupply` and `ConstantProduct2::calcReserve` should exhibit an accurate inverse relationship (`calcLpTokenSupply(calcReserves(reserves, LP)) == LP`). In practice, all calculations come with rounding to some extent, and this relationship is not possible so long as the two functions are implemented separately.

Recommended Mitigation:

1. Add a rounding direction flag parameter to the `IWellFunction::calcReserve` and `IWellFunction::calcLpTokenSupply` functions. At all invocations of these functions, apply the correct rounding direction such that the transaction is processed favorably to the protocol.
2. Handle the last token separately in `Well::removeLiquidity`. In `Well::getRemoveLiquidityOut`, calculate the output token amount for the first `n-1` tokens as usual but use an approach similar to `Well::getRemoveLiquidityOneTokenOut` for the last token. This will help to have the invariant hold. We understand that another finding suggested adding a dedicated function to calculate the output token amount at the `IWellFunction` level. If the issue is mitigated that way, consider adding another check to ensure the invariant holds.
3. As we explained above, it is practically almost impossible to make the invariant `totalSupply() == calcLpTokenSupply(reserves)` hold strictly. We recommend a slightly looser invariant `totalSupply() >= calcLpTokenSupply(reserves)`, which can be interpreted as "the Well function underestimates the LP supply". Otherwise, we suggest adding another variable, `totalCalcSupply`, which will always reflect the calculated LP supply from the current `reserves`. This new variable can be used with `totalSupply()` together while deciding the output token amount. For example, in `Well::_getRemoveLiquidityOneTokenOut`, we can calculate `lpAmountInCalc = lpAmountIn * totalCalcSupply / totalSupply()` and use that in the function `_calcReserve`. This will be helpful because the discrepancy is reflected in the calculation.

Please note that `well.getReserves()[i] == wellFunction.calcReserve(i)` is also supposed to hold. Because of the rounding in the calculation, we understand this invariant might be too strict and similar mitigation is recommended. Below is a test to show that the above *invariant* can be broken.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.17;

import {IERC20} from "test/TestHelper.sol";
import {IWellFunction} from "src/functions/ConstantProduct2.sol";
import {LiquidityHelper} from "test/LiquidityHelper.sol";
import "forge-std/Test.sol";
import {MockToken} from "mocks/tokens/MockToken.sol";

contract ReservesMathFunctionCalcReservesFail is LiquidityHelper {
    function setUp() public {
        setupWell(2);
    }

    function testReservesMismatchFail() public {
        // perform initial check
        uint[] memory reserves = well.getReserves();

        uint reserve0 =
            IWellFunction(wellFunction.target).calcReserve(reserves, 0, well.totalSupply(),
            ↪  wellFunction.data);
        uint reserve1 =
            IWellFunction(wellFunction.target).calcReserve(reserves, 1, well.totalSupply(),
            ↪  wellFunction.data);

        assertEq(reserves[0], reserve0);
```

```
        assertEq(reserves[1], reserve1);

        // swap
        address msgSender = address(2);
        changePrank(msgSender);
        uint tokenInIndex = 0;
        uint tokenOutIndex = 1;
        uint amountIn = 59_534_739_918_926_377_591_512_171_513;

        IERC20[] memory mockTokens = well.tokens();
        MockToken(address(mockTokens[tokenInIndex])).mint(msgSender, amountIn);
        // approve the well
        mockTokens[tokenInIndex].approve(address(well), amountIn);
        uint minAmountOut = well.getSwapOut(mockTokens[tokenInIndex], mockTokens[tokenOutIndex],
        ↪  amountIn);
        well.swapFrom(
            mockTokens[tokenInIndex], mockTokens[tokenOutIndex], amountIn, minAmountOut, msgSender,
            ↪  block.timestamp
        );

        // perform check again
        reserves = well.getReserves();

        reserve0 = IWellFunction(wellFunction.target).calcReserve(reserves, 0, well.totalSupply(),
        ↪  wellFunction.data);
        reserve1 = IWellFunction(wellFunction.target).calcReserve(reserves, 1, well.totalSupply(),
        ↪  wellFunction.data);

        assertEq(reserves[0], reserve0);
        assertEq(reserves[1], reserve1);
    }
}
```

Beanstalk:

1. `ConstantProduct2::_calcReserve` now rounds up instead of to the nearest number. `ConstantProduct2::_calcLpTokenSupply` still rounds down. The rationale is below. Fixed in commit 876da9a.

   No rounding mode is necessary because `calcReserve` and `calcLpTokenSupply` should round the same direction in all cases:

   1. All instances of `calcReserve` should round up:
      - All output `amountIn` values are calculated by subtracting `_calcReserve(...) - reserveBefore`. The Well wants to overestimate `amountIn` to guarantee more tokens on the margin are sent to the Well. Thus, `_calcReserve` should round up. (See `swapTo`)
      - All output `amountOut` values are calculated by subtracting `reserveBefore - _calcReserve(...)`. The Well wants to underestimate `amountOut` so that less tokens are removed from the Well on the margin. Thus, `_calcReserve` should round up again. (See `swapFrom`, `shift`, `removeLiquidityOneToken`)
   2. All instances of `calcLpTokenSupply` should round down:
      - All `lpAmountIn` values are calculated by subtracting `totalSupply() - _calcLpTokenSupply(...)`. `lpAmountIn` should be overestimated and thus `_calcLpTokenSupply` should round down. (See `removeLiquidityImbalanced`)
      - All `lpAmountOut` values are calculated by subtracting `_calcLpTokenSupply(...) - totalSupply()`. `lpAmountOut` values should be underestimated and thus `_calcLpTokenSupply`

should round down.

Given no rounding mode is necessary–It is in fact up to the corresponding Well Function to decide how to round. The above rounding methods are only a suggestion and each Well Function is free to round however it wants.

2. The remediation for [H-03] added a `calcLPTokenUnderlying` function to `IWellFunction` to determine how much of each amount to remove when removing LP tokens. As a consequence, the Well Function itself now determines how many tokens to remove when LP is removed instead of the Well. Thus, the Well Function is free to define how to handle the `removeLiquidity` calculation and can ensure that it doesn't break its own defined invariant.

   Documentation has been added here to indicate that a valid implementation of `calcLPTokenUnderlying` should never break a Well's invariant.

   Also, updated the `test_removeLiquidity_fuzz` test to include a check to test that the invariant holds after the `removeLiquidity` call here.

   Given that users should verify that a Well has a valid Well Function before interacting with a Well and protocols using Wells should keep a list of verified Well Functions, adding a check would just serve as an extra precaution in the case that an invalid Well Function is verified.,

   Adding the following check to `removeLiquidity`:

   ```
   require(lpTokenSupply - lpAmountIn <= _calcLpTokenSupply(wellFunction(), reserves), "Well
   ↪  Function Invalid");
   ```

   increases the gas cost by 3,940. Given that the check only serves as an extra precaution, it doesn't seem like it is worth the extra gas cost to add. Therefore, the check has not been added.

3. The invariant that the Well is maintaining is `_calcLpTokenSupply(...) >= totalSupply()`. In order words, the Well issues fewer Well LP Tokens at the margin. This is to ensure that Well LP Tokens at least maintain their underlying value and don't get diluted.

   In order to help verify this, a `checkInvariant` function has been added to `TestHelper` which reverts if the above invariant does not hold. The `checkInvariant` function has been added to the majority of tests including `swapFrom`, `swapTo`, `removeLiquidity`, `removeLiquidityImbalanced`, `removeLiquidityOneToken`, and `addLiquidity`. Added in commit 876da9a.

Cyfrin: Acknowledged and validated changes mitigate the original issue.

6.1.2 Each Well is responsible for ensuring that an `update` call cannot be made with a reserve of 0

Description: The current implementation of `GeoEmaAndCumSmaPump` assumes each well will call `update()` with non-zero reserves, as commented at the beginning of the file:

```
/**
 * @title GeoEmaAndCumSmaPump
 * @author Publius
 * @notice Stores a geometric EMA and cumulative geometric SMA for each reserve.
 * @dev A Pump designed for use in Beanstalk with 2 tokens.
 *
 * This Pump has 3 main features:
 *  1. Multi-block MEV resistence reserves
 *  2. MEV-resistant Geometric EMA intended for instantaneous reserve queries
 *  3. MEV-resistant Cumulative Geometric intended for SMA reserve queries
 *
 * Note: If an `update` call is made with a reserve of 0, the Geometric mean oracles will be set to 0.
 * Each Well is responsible for ensuring that an `update` call cannot be made with a reserve of 0.
 */
```

However, there is no actual requirement in `Well` to enforce pump updates with valid reserve values. Given that `GeoEmaAndCumSmaPump` restricts values to a minimum of 1 to prevent issues with the geometric mean, that the TWA values are not truly representative of the reserves in the Well, we believe it is worse than reverting in this case, although a `ConstantProduct2` Well can have zero reserves for either token via valid transactions.

```
GeoEmaAndCumSmaPump.sol
103:         for (uint i; i < length; ++i) {
104:             // Use a minimum of 1 for reserve. Geometric means will be set to 0 if a reserve is 0.
105:             b.lastReserves[i] =
106:                 _capReserve(b.lastReserves[i], (reserves[i] > 0 ? reserves[i] :
↪  1).fromUIntToLog2(), blocksPassed);
107:             b.emaReserves[i] =
↪  b.lastReserves[i].mul((ABDKMathQuad.ONE.sub(aN))).add(b.emaReserves[i].mul(aN));
108:             b.cumulativeReserves[i] =
↪  b.cumulativeReserves[i].add(b.lastReserves[i].mul(deltaTimestampBytes));
109:         }
```

Impact: Updating pumps with zero reserve values can lead to the distortion of critical states likely to be utilized for price oracles. Given that the issue is exploitable through valid transactions, we assess the severity as HIGH. It is crucial to note that attackers can exploit this vulnerability to manipulate the price oracle.

Proof of Concept: The test below shows that it is possible for reserves to be zero through valid transactions and updating pumps do not revert.

```
function testUpdateCalledWithZero() public {
    address msgSender = 0x83a740c22a319FBEe5F2FaD0E8Cd0053dC711a1A;
    changePrank(msgSender);
    IERC20[] memory mockTokens = well.tokens();

    // add liquidity 1 on each side
    uint amount = 1;
    MockToken(address(mockTokens[0])).mint(msgSender, 1);
    MockToken(address(mockTokens[1])).mint(msgSender, 1);
    MockToken(address(mockTokens[0])).approve(address(well), amount);
    MockToken(address(mockTokens[1])).approve(address(well), amount);
    uint[] memory tokenAmountsIn = new uint[](2);
    tokenAmountsIn[0] = amount;
    tokenAmountsIn[1] = amount;
    uint minLpAmountOut = well.getAddLiquidityOut(tokenAmountsIn);
    well.addLiquidity(
        tokenAmountsIn,
        minLpAmountOut,
        msgSender,
        block.timestamp
    );

    // swaFromFeeOnTransfer from token1 to token0
    msgSender = 0xfFfFFffFfffffFFffFffFFFFFFfFFFfFfFFfFfD;
    changePrank(msgSender);
    amount = 79_228_162_514_264_337_593_543_950_334;
    MockToken(address(mockTokens[1])).mint(msgSender, amount);
    MockToken(address(mockTokens[1])).approve(address(well), amount);
    uint minAmountOut = well.getSwapOut(
        mockTokens[1],
        mockTokens[0],
        amount
    );

    well.swapFromFeeOnTransfer(
        mockTokens[1],
        mockTokens[0],
```

```
        amount,
        minAmountOut,
        msgSender,
        block.timestamp
    );
    increaseTime(120);

    // remove liquidity one token
    msgSender = address(this);
    changePrank(msgSender);
    amount = 999_999_999_999_999_999_999_999_999;
    uint minTokenAmountOut = well.getRemoveLiquidityOneTokenOut(
        amount,
        mockTokens[1]
    );
    well.removeLiquidityOneToken(
        amount,
        mockTokens[1],
        minTokenAmountOut,
        msgSender,
        block.timestamp
    );

    msgSender = address(12_345_678);
    changePrank(msgSender);

    vm.warp(block.timestamp + 1);
    amount = 1;
    MockToken(address(mockTokens[0])).mint(msgSender, amount);
    MockToken(address(mockTokens[0])).approve(address(well), amount);
    uint amountOut = well.getSwapOut(mockTokens[0], mockTokens[1], amount);

    uint[] memory reserves = well.getReserves();
    assertEq(reserves[1], 0);

    // we are calling `_update` with reserves of 0, this should fail
    well.swapFrom(
        mockTokens[0],
        mockTokens[1],
        amount,
        amountOut,
        msgSender,
        block.timestamp
    );
}
```

Recommended Mitigation: Revert the pump updates if they are called with zero reserve values.

Beanstalk: Have decided not to implement the recommended remediation. This is due to several factors. First, there have been two changes made to the pump structure:

1) Have the Well not revert on Pump failure (commit 2459058). This was implemented to protect the Well against Pump failure. Without this change, if the Pump breaks for some reason, the liquidity will be permanently locked as all Well actions will fail.
2) Don't initialize a Pump if any of the reserves are zero. This is to prevent the EMA and last balances from being initialized to zero. Instead, the Pump will just return and wait until it receives non-zero balances to initialize.

Due to 1), if the recommended remediation was implemented, then in the case where a balance of zero was passed in, the Well will continue to function, but the Pump will not be updated. Say a balance in the Well is 1e6, it

is then set to 0 for an hour and then set back to 1e6. Because the Pump failed when the Well tried to update the Pump with the zero balance, it was not updated at all. Thus, the Pump will assume that the balance was 1e6 the whole time and not set to zero, which implies that the Pump is not an accurate measure of historical balances and could be manipulated (assuming the Well Function could use a balance of 0).

In addition, the difference between a balance of 1 and 0 is quite small. First, in neither case is the Well acting as a reliable source of price discovery. If there is only 1 micro-unit of an ERC-20 token in a Well, then there is essentially no bid to sell any more of that asset given the value of 1 micro-unit of an ERC-20 token is infinitesimal. For this reason, any protocol using the Pump should ensure that there is a sufficient balance of both ERC-20 tokens in the Well to ensure that it is an accurate source of price discovery. Therefore, the consequence of using a balance of 1 when the Pump intakes a balance of 0 should be minimal.

Cyfrin: Acknowledged.

6.1.3 `removeLiquidity` logic is not correct for generalized Well functions other than ConstantProduct

Description: The protocol intends to provide a generalized framework for constant-function AMM liquidity pools where various Well functions can be used. Currently, only the constant-product type Well function is defined, but we understand that more general Well functions are intended to be supported.

The current implementation of `Well::removeLiquidity` and `Well::getRemoveLiquidityOut` assumes linearity while getting the output token amount from the LP token amount to withdraw. This holds well for the constant product type, as seen below. If we denote the total supply of LP tokens as L, the reserve values for the two tokens as x, y, and the invariant is $L^2=4xy$ for `ConstantProduct2`. When removing liquidity of amount l, the output amounts are calculated as $\Delta x=\frac{l}{L}x, \Delta y=\frac{l}{L}y$. It is straightforward to verify that the invariant still holds after withdrawal, i.e., $(L-l)^2=(x-\Delta x)(y-\Delta y)$.

But in general, this kind of *linearity* is not guaranteed to hold.

Recently, non-linear (quadratic) function AMMs have been introduced by some new protocols (see Numoen). If we use this kind of Well function, the current calculation of `tokenAmountsOut` will break the Well's invariant.

For your information, the Numoen protocol checks the protocol's invariant (the constant function itself) after every transaction.

Impact: The current `Well::removeLiquidity` logic assumes a specific condition on the Well function (linearity in some sense). This limits the generalization of the protocol as opposed to its original purpose. Given that this will lead to loss of funds for the liquidity providers for general Well functions, we evaluate the severity to HIGH.

Proof of Concept: We wrote a test case with the quadratic Well function used by Numoen.

```
// QuadraticWell.sol

/**
 * SPDX-License-Identifier: MIT
 **/

pragma solidity ^0.8.17;

import "src/interfaces/IWellFunction.sol";
import "src/libraries/LibMath.sol";

contract QuadraticWell is IWellFunction {
    using LibMath for uint;

    uint constant PRECISION = 1e18; //@audit-info assume 1:1 upperbound for this well
    uint constant PRICE_BOUND = 1e18;

    /// @dev s = b_0 - (p_1^2 - b_1/2)^2
    function calcLpTokenSupply(
        uint[] calldata reserves,
        bytes calldata
    ) external override pure returns (uint lpTokenSupply) {
```

```
        uint delta = PRICE_BOUND - reserves[1] / 2;
        lpTokenSupply = reserves[0] - delta*delta/PRECISION ;
    }

    /// @dev b_0 = s + (p_1^2 - b_1/2)^2
    /// @dev b_1 = (p_1^2 - (b_0 - s)^(1/2))*2
    function calcReserve(
        uint[] calldata reserves,
        uint j,
        uint lpTokenSupply,
        bytes calldata
    ) external override pure returns (uint reserve) {

        if(j == 0)
        {
            uint delta = PRICE_BOUND*PRICE_BOUND - PRECISION*reserves[1]/2;
            return lpTokenSupply + delta*delta /PRECISION/PRECISION/PRECISION;
        }
        else {
            uint delta = (reserves[0] - lpTokenSupply)*PRECISION;
            return (PRICE_BOUND*PRICE_BOUND - delta.sqrt()*PRECISION)*2/PRECISION;
        }
    }

    function name() external override pure returns (string memory) {
        return "QuadraticWell";
    }

    function symbol() external override pure returns (string memory) {
        return "QW";
    }
}

// NOTE: Put in Exploit.t.sol
function test_exploitQuadraticWellAddRemoveLiquidity() public {
    MockQuadraticWell quadraticWell = new MockQuadraticWell();
    Call memory _wellFunction = Call(address(quadraticWell), "");
    Well well2 = Well(auger.bore("Well2", "WELL2", tokens, _wellFunction, pumps));

    approveMaxTokens(user, address(well2));
    uint[] memory amounts = new uint[](tokens.length);
    changePrank(user);

    // initial status 1:1
    amounts[0] = 1e18;
    amounts[1] = 1e18;
    well2.addLiquidity(amounts, 0, user); // state: [1 ether, 1 ether, 0.75 ether]

    Balances memory userBalances1 = getBalances(user, well2);
    uint[] memory userBalances = new uint[](3);
    userBalances[0] = userBalances1.tokens[0];
    userBalances[1] = userBalances1.tokens[1];
    userBalances[2] = userBalances1.lp;
    Balances memory wellBalances1 = getBalances(address(well2), well2);
    uint[] memory wellBalances = new uint[](3);
    wellBalances[0] = wellBalances1.tokens[0];
    wellBalances[1] = wellBalances1.tokens[1];
    wellBalances[2] = wellBalances1.lpSupply;
    amounts[0] = wellBalances[0];
    amounts[1] = wellBalances[1];

    emit log_named_array("userBalances1", userBalances);
```

```
    emit log_named_array("wellBalances1", wellBalances);
    emit log_named_int("invariant", quadraticWell.wellInvariant(wellBalances[2], amounts));

    // addLiquidity
    amounts[0] = 2e18;
    amounts[1] = 1e18;
    well2.addLiquidity(amounts, 0, user); // state: [3 ether, 2 ether, 3 ether]

    Balances memory userBalances2 = getBalances(user, well2);
    userBalances[0] = userBalances2.tokens[0];
    userBalances[1] = userBalances2.tokens[1];
    userBalances[2] = userBalances2.lp;
    Balances memory wellBalances2 = getBalances(address(well2), well2);
    wellBalances[0] = wellBalances2.tokens[0];
    wellBalances[1] = wellBalances2.tokens[1];
    wellBalances[2] = wellBalances2.lpSupply;
    amounts[0] = wellBalances[0];
    amounts[1] = wellBalances[1];

    emit log_named_array("userBalances2", userBalances);
    emit log_named_array("wellBalances2", wellBalances);
    emit log_named_int("invariant", quadraticWell.wellInvariant(wellBalances[2], amounts));

    // removeLiquidity
    amounts[0] = 0;
    amounts[1] = 0;
    well2.removeLiquidity(userBalances[2], amounts, user);

    Balances memory userBalances3 = getBalances(user, well2);
    userBalances[0] = userBalances3.tokens[0];
    userBalances[1] = userBalances3.tokens[1];
    userBalances[2] = userBalances3.lp;
    Balances memory wellBalances3 = getBalances(address(well2), well2);
    wellBalances[0] = wellBalances3.tokens[0];
    wellBalances[1] = wellBalances3.tokens[1];
    wellBalances[2] = wellBalances3.lpSupply;
    amounts[0] = wellBalances[0];
    amounts[1] = wellBalances[1];

    emit log_named_array("userBalances3", userBalances);
    emit log_named_array("wellBalances3", wellBalances);
    emit log_named_int("invariant", quadraticWell.wellInvariant(wellBalances[2], amounts)); //
    ↪  @audit-info well's invariant is broken via normal removeLiquidity
}
```

The output is shown below. We calculated the `invariant` of the Well after transactions, and while it is supposed to stay at zero, it is broken after removing liquidity. Note that the invariant stayed zero on adding liquidity because the protocol explicitly calculates the resulting liquidity token supply using the Well function. However, the output amounts are calculated in a fixed manner when removing liquidity without using the Well function, which breaks the invariant.

```
forge test -vv --match-test test_exploitQuadraticWellAddRemoveLiquidity

[PASS] test_exploitQuadraticWellAddRemoveLiquidity() (gas: 4462244)
Logs:
  userBalances1: [99900000000000000000, 99900000000000000000, 750000000000000000]
  wellBalances1: [1000000000000000000, 1000000000000000000, 750000000000000000]
  invariant: 0
  userBalances2: [99700000000000000000, 99800000000000000000, 3000000000000000000]
  wellBalances2: [3000000000000000000, 2000000000000000000, 3000000000000000000]
  invariant: 0
  userBalances3: [100000000000000000000, 100000000000000000000, 0]
  wellBalances3: [0, 0, 0]
  invariant: 1000000000000000000

Test result: ok. 1 passed; 0 failed; finished in 5.14ms
```

Recommended Mitigation: We believe it is impossible to cover all kinds of Well functions without adding some additional functions in the interface `IWellFunction`. We recommend adding a new function in the `IWellFunction` interface, possibly in the form of `function calcWithdrawFromLp(uint lpTokenToBurn) returns (uint reserve)`. The output token amount can be calculated using the newly added function.

Beanstalk: Added a `calcLPTokenUnderlying` function to `IWellFunction`. It returns the amount of each reserve token underlying a given amount of LP tokens. It is used to determine how many underlying tokens to send to a user when they remove liquidity using the `removeLiquidity` function. Fixed in commit 5271e9a.

Cyfrin: Acknowledged.

6.1.4 Read-only reentrancy

Description: The current implementation is vulnerable to read-only reentrancy, especially in Wells::removeLiquidity. The implementation does not strictly follow the Checks-Effects-Interactions (CEI) pattern as it is setting the new reserve values after sending out the tokens. This is not an immediate risk to the protocol itself due to the `nonReentrant` modifier, but this is still vulnerable to read-only reentrancy.

Malicious attackers and unsuspecting ecosystem participants can deploy Wells with ERC-777 tokens (which have a callback that can take control) and exploit this vulnerability. This will lead to critical vulnerabilities given that the Wells are to be extended with price functions as defined by pumps - third-party protocols that integrate these on-chain oracles will be at risk.

Pumps are updated before token transfers; however, reserves are only set after. Therefore, pump functions will likely be incorrect on a re-entrant read-only call if `IWell(well).getReserves()` is called but reserves have not been correctly updated. The implementation of `GeoEmaAndCumSmaPump` appears not to be vulnerable, but given that each pump can choose its approach for recording a well's reserves over time, this remains a possible attack vector.

Impact: Although this is not an immediate risk to the protocol itself, read-only re-entrancy can lead to critical issues, so we evaluate the severity as HIGH.

Proof of Concept: We wrote a test case to show the existing read-only reentrancy.

```
// MockCallbackRecipient.sol

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import {console} from "forge-std/Test.sol";

contract MockCallbackRecipient {
    fallback() external payable {
        console.log("here");
        (bool success, bytes memory result) = msg.sender.call(abi.encodeWithSignature("getReserves()"));
```

```
        if (success) {
            uint256[] memory reserves = abi.decode(result, (uint256[]));
            console.log("read-only-reentrancy beforeTokenTransfer reserves[0]: %s", reserves[0]);
            console.log("read-only-reentrancy beforeTokenTransfer reserves[1]: %s", reserves[1]);
        }
    }
}

// NOTE: Put in Exploit.t.sol
function test_exploitReadOnlyReentrancyRemoveLiquidityCallbackToken() public {
    IERC20 callbackToken = IERC20(new MockCallbackToken("CallbackToken", "CBTKN", 18));
    MockToken(address(callbackToken)).mint(user, 1000e18);
    IERC20[] memory _tokens = new IERC20[](2);
    _tokens[0] = callbackToken;
    _tokens[1] = tokens[1];

    vm.stopPrank();
    Well well2 = Well(auger.bore("Well2", "WELL2", _tokens, wellFunction, pumps));
    approveMaxTokens(user, address(well2));

    uint[] memory amounts = new uint[](2);
    amounts[0] = 100 * 1e18;
    amounts[1] = 100 * 1e18;

    changePrank(user);
    callbackToken.approve(address(well2), type(uint).max);
    uint256 lpAmountOut = well2.addLiquidity(amounts, 0, user);

    well2.removeLiquidity(lpAmountOut, amounts, user);
}
```

The output is shown below.

```
forge test -vv --match-test test_exploitReadOnlyReentrancyRemoveLiquidityCallbackToken

[PASS] test_exploitReadOnlyReentrancyRemoveLiquidityCallbackToken() (gas: 5290876)
Logs:
  read-only-reentrancy beforeTokenTransfer reserves[0]: 0
  read-only-reentrancy beforeTokenTransfer reserves[1]: 0
  read-only-reentrancy afterTokenTransfer reserves[0]: 0
  read-only-reentrancy afterTokenTransfer reserves[1]: 0
  read-only-reentrancy beforeTokenTransfer reserves[0]: 100000000000000000000
  read-only-reentrancy beforeTokenTransfer reserves[1]: 100000000000000000000
  read-only-reentrancy afterTokenTransfer reserves[0]: 100000000000000000000
  read-only-reentrancy afterTokenTransfer reserves[1]: 100000000000000000000

Test result: ok. 1 passed; 0 failed; finished in 3.66ms
```

Recommended Mitigation: Implement the CEI pattern in relevant functions by updating reserves before making external calls. For example, the function `Well::removeLiquidity` can be modified shown below.

```
function removeLiquidity(
    uint lpAmountIn,
    uint[] calldata minTokenAmountsOut,
    address recipient
) external nonReentrant returns (uint[] memory tokenAmountsOut) {
    IERC20[] memory _tokens = tokens();
    uint[] memory reserves = _updatePumps(_tokens.length);
    uint lpTokenSupply = totalSupply();

    tokenAmountsOut = new uint[](_tokens.length);
    _burn(msg.sender, lpAmountIn);

    _setReserves(reserves); // @audit CEI pattern

    for (uint i; i < _tokens.length; ++i) {
        tokenAmountsOut[i] = (lpAmountIn * reserves[i]) / lpTokenSupply;
        require(
            tokenAmountsOut[i] >= minTokenAmountsOut[i],
            "Well: slippage"
        );
        _tokens[i].safeTransfer(recipient, tokenAmountsOut[i]);
        reserves[i] = reserves[i] - tokenAmountsOut[i];
    }

    emit RemoveLiquidity(lpAmountIn, tokenAmountsOut);
}
```

Beanstalk: Added a check to the `getReserves()` function that reverts if the Reentrancy guard has been entered. This prevents anyone from calling `getReserves()` while executing a function in the Well. Fixed in commit fcbf04a.

Cyfrin: Acknowledged.

6.2 Medium Risk

6.2.1 Should ensure uniqueness of the tokens of Wells

Description: The current implementation does not enforce uniqueness in the tokens of Wells. Anyone can call `Aquifer::boreWell()` with malicious `Well` implementation to set up a trap for victims. Through communication with the protocol team, it is understood that all Wells are considered *guilty until proven innocent*. But it is still desirable to verify the Well on `Aquifier::boreWell` and prevent deployments of malicious Wells. It is also strongly recommended to prohibit changing `tokens()` after the deployment.

If a Well has duplicate tokens, an attack path shown below exists, and there can be more.

Impact: While we assume users will be warned explicitly about malicious Wells and not likely to interact with invalid Wells, we evaluate the severity to MEDIUM.

Proof of Concept: Let us say tokens[0]=tokens[1]. An honest LP calls `addLiquidity([1 ether,1 ether], 200 ether, address)`, and the reserves will be (1 ether, 1 ether). But anyone can call `skim()` and take `1 ether` out. This is because `skimAmounts` relies on the `balanceOf()`, which will return `2 ether` for the first loop.

```
function skim(address recipient) external nonReentrant returns (uint[] memory skimAmounts) {
    IERC20[] memory _tokens = tokens();
    uint[] memory reserves = _getReserves(_tokens.length);
    skimAmounts = new uint[](_tokens.length);
    for (uint i; i < _tokens.length; ++i) {
        skimAmounts[i] = _tokens[i].balanceOf(address(this)) - reserves[i];
        if (skimAmounts[i] > 0) {
            _tokens[i].safeTransfer(recipient, skimAmounts[i]);
        }
    }
}
```

Recommended Mitigation:

- Do not allow changing the Well tokens once deployed.
- Require the uniqueness of the tokens during `boreWell()`.

Beanstalk: Fixed in commit f10e05a.

Cyfrin: Acknowledged.

6.2.2 `LibLastReserveBytes::storeLastReserves` has no check for reserves being too large

Description: After every liquidity event & swap, the `IPump::update()is called. To update the pump, the`LibLastReserveBytes::storeLastReserves`function is used. This packs the reserve data into`bytes32‘ slots in storage. A slot is then broken down into the following components:

- 1 byte for reserves array length
- 5 bytes for `timestamp`
- 16 bytes for each reserve balance

This adds to 22 bytes total, but the function also attempts to pack the second reserve balance in the `bytes32` object. This would mean the `bytes32` would need 38 bytes total:

`1(length) + 5(timestamp) + 16(reserve balance 1) + 16(reserve balance 2) = 38 bytes`

To fit all this data into the `bytes32`, the function cuts off the last few bytes of the reserve balances using shift, as shown below.

```
src\libraries\LibLastReserveBytes.sol
21:     uint8 n = uint8(reserves.length);
22:     if (n == 1) {
23:         assembly {
24:             sstore(slot, or(or(shl(208, lastTimestamp), shl(248, n)), shl(104, shr(152,
↪  mload(add(reserves, 32))))))
25:         }
26:         return;
27:     }
28:     assembly {
29:         sstore(
30:             slot,
31:             or(
32:                 or(shl(208, lastTimestamp), shl(248, n)),
33:                 or(shl(104, shr(152, mload(add(reserves, 32)))), shr(152, mload(add(reserves, 64))))
34:             )
35:         )
36:         // slot := add(slot, 32)
37:     }
```

So if the amount being stored is too large, the actual stored value will be different than what was expected to be stored.

On the other hand, the `LibBytes.sol` does seem to have a check:

```
require(reserves[0] <= type(uint128).max, "ByteStorage: too large");
```

The `_setReserves` function calls this library after every reserve update in the well. So in practice, with the currently implemented wells & pumps, this check would cause a revert.

However, a well that is implemented without this check could additionally trigger the pumps to cut off reserve data, meaning prices would be incorrect.

Impact: While we assume users will be explicitly warned about malicious Wells and are unlikely to interact with invalid Wells, we assess the severity to be MEDIUM.

Proof of Concept:

```
function testStoreAndReadTwo() public {
    uint40 lastTimeStamp = 12345363;
    bytes16[] memory reserves = new bytes16[](2);
    reserves[0] = 0xffffffffffffffffffffffffffffffff; // This is too big!
    reserves[1] = 0x11111111111111111111111100000000;
    RESERVES_STORAGE_SLOT.storeLastReserves(lastTimeStamp, reserves);
    (
        uint8 n,
        uint40 _lastTimeStamp,
        bytes16[] memory _reserves
    ) = RESERVES_STORAGE_SLOT.readLastReserves();
    assertEq(2, n);
    assertEq(lastTimeStamp, _lastTimeStamp);
    assertEq(reserves[0], _reserves[0]); // This will fail
    assertEq(reserves[1], _reserves[1]);
    assertEq(reserves.length, _reserves.length);
}
```

Recommended Mitigation: We recommend adding a check on the size of reserves in `LibLastReseveBytes`.

Additionally, it is recommended to add comments to `LibLastReseveBytes` to inform users about the invariants of the system and how the max size of reserves should be equal to the max size of a `bytes16` and not a `uint256`.

Beanstalk: Because the Pump packs 2 last reserve values in the form of `bytes16` quadruple precision floating point values and a `uint40` timestamp into the same slot, there is a loss of precision on last reserve values. Each last reserve value only has precision of ~27 decimals instead of the expected ~34 decimals.

Given that the last reserves are only used to determine the cap on reserve updates and that the 27 decimals that are preserved are the most significant decimals, the impact due to this is minimal. The cap is only used to prevent the effect of manipulation, and is arbitrarily set. It is also never evaluated by external protocols. Finally, 27 decimal precision is still quite significant $\rightarrow$ If would need to be about 1,000,000,000,000,000,000,000,000,000 tokens in the pool for there to be an error of 1.

Cyfrin: Acknowledged.

6.3 Low Risk

6.3.1 TWAP is incorrect when only 1 update has occurred

Description: If there has only been a single update to the pump, `GeoEmaAndCumSmaPump::readTwaReserves` will return an incorrect value.

Impact: Given this affects only one oracle update when the pump is still ramping up, we evaluate the severity to LOW.

Proof of Concept:

```
// NOTE: place in `Pump.Update.t.sol`
function testTWAReservesIsWrong() public {
    increaseTime(12); // increase 12 seconds

    bytes memory startCumulativeReserves = pump.readCumulativeReserves(
        address(mWell)
    );
    uint256 lastTimestamp = block.timestamp;
    increaseTime(120); // increase 120 seconds aka 10 blocks

    (uint[] memory twaReserves, ) = pump.readTwaReserves(
        address(mWell),
        startCumulativeReserves,
        lastTimestamp
    );

    assertApproxEqAbs(twaReserves[0], 1e6, 1);
    assertApproxEqAbs(twaReserves[1], 2e6, 1);

    vm.prank(user);
    // gonna double it
    // so our TWAP should now be ~1.5 & 3
    b[0] = 2e6;
    b[1] = 4e6;
    mWell.update(address(pump), b, new bytes(0));

    increaseTime(120); // increase 12 seconds aka 10 blocks
    (twaReserves, ) = pump.readTwaReserves(
        address(mWell),
        startCumulativeReserves,
        lastTimestamp
    );

    // the reserves of b[0]:
    // - 1e6 * 10 blocks
    // - 2e6 * 10 blocks
    // average reserves over 20 blocks:
    // - 15e5
    // assertApproxEqAbs(twaReserves[0], 1.5e5, 1);
    // instead, we get:
    // b[0] = 2070529

    // the reserves of b[1]:
    // - 2e6 * 10 blocks
    // - 4e6 * 10 blocks
    // average reserves over 20 blocks:
    // - 3e6
    assertApproxEqAbs(twaReserves[1], 3e6, 1);
    // instead, we get:
    // b[1] = 4141059
}
```

Recommended Mitigation: Disallow reading from the pump with fewer than 2 updates.

Beanstalk: This is due to a bug in `MockReserveWell`. Before the Well would make the Pump update call with the new reserves. However, the Well should update the Pump with the previous reserves.

This issue with `MockReserveWell` has been fixed here.

Upon running the test now, you will see that `twaReserve[0] = 1414213`. This is expected as `sqrt(1e6 * 2e6) = 1414213`. Also, `twaReserve[1] = 2828427`. This is also expected as `sqrt(2e6 * 4e6) = 2828427`.

Cyfrin: Acknowledged.

6.3.2 Lack of validation for `A` in `GeoEmaAndCumSmaPump::constructor`

Description: In the `GeoEmaAndCumSmaPump::constructor`, the EMA parameter (α) `A` is initialized to `_A`. This parameter is supposed to be less than 1 otherwise `GeoEmaAndCumSmaPump::update` will revert due to underflow.

Impact: Given this initialization is done by the deployer on initialization, we evaluate the severity to LOW.

Recommended Mitigation: Require the initialization value `_A` to be less than `ABDKMathQuad.ONE`.

Beanstalk: Fixed in commit e834f9f.

Cyfrin: Acknowledged.

6.3.3 Incorrect sload in LibBytes

Description: The function `storeUint128` in `LibBytes` intends to pack uint128 `reserves` starting at the given slot but will overwrite the final slot if storing an odd number of reserves. It is currently only ever called in `Well::_setReserves` which takes as input the result of `Well::_updatePumps` which itself always takes `_tokens.length` as an argument. Hence, in the case of an odd number of tokens, the final 128 bits in the slot are never accessed, regardless of the error. However, there may be a case in which the library is used by other implementations, setting a variable number of reserves at any one time rather than always acting on the length of tokens, which may inadvertently overwrite the final reserve to zero.

Impact: Given assets are not directly at risk, we evaluate the severity to LOW.

Proof of Concept: The following test case demonstrates this issue more clearly:

```
// NOTE: Add to LibBytes.t.sol
function test_exploitStoreAndRead() public {
    // Write to storage slot to demonstrate overwriting existing values
    // In this case, 420 will be stored in the lower 128 bits of the last slot
    bytes32 slot = RESERVES_STORAGE_SLOT;
    uint256 maxI = (NUM_RESERVES_MAX - 1) / 2;
    uint256 storeValue = 420;
    assembly {
        sstore(add(slot, mul(maxI, 32)), storeValue)
    }

    // Read reserves and assert the final reserve is 420
    uint[] memory reservesBefore = LibBytes.readUint128(RESERVES_STORAGE_SLOT, NUM_RESERVES_MAX);
    emit log_named_array("reservesBefore", reservesBefore);

    // Set up reserves to store, but only up to NUM_RESERVES_MAX - 1 as we have already stored a value
    ↪  in the last 128 bits of the last slot
    uint[] memory reserves = new uint[](NUM_RESERVES_MAX - 1);
    for (uint i = 1; i < NUM_RESERVES_MAX; i++) {
        reserves[i-1] = i;
    }
```

```
    // Log the last reserve before the store, perhaps from other implementations which don't always act
    ↪  on the entire reserves length
    uint256 t;
    assembly {
        t := shr(128, shl(128, sload(add(slot, mul(maxI, 32)))))
    }
    emit log_named_uint("final slot, lower 128 bits before", t);

    // Store reserves
    LibBytes.storeUint128(RESERVES_STORAGE_SLOT, reserves);

    // Re-read reserves and compare
    uint[] memory reserves2 = LibBytes.readUint128(RESERVES_STORAGE_SLOT, NUM_RESERVES_MAX);

    emit log_named_array("reserves", reserves);
    emit log_named_array("reserves2", reserves2);

    // But wait, what about the last reserve
    assembly {
        t := shr(128, shl(128, sload(add(slot, mul(maxI, 32)))))
    }

    // Turns out it was overwritten by the last store as it calculates the sload incorrectly
    emit log_named_uint("final slot, lower 128 bits after", t);
}
```

Output before mitigation:

```
reservesBefore: [0, 0, 0, 0, 0, 0, 0, 420]
final slot, lower 128 bits before: 420
reserves: [1, 2, 3, 4, 5, 6, 7]
reserves2: [1, 2, 3, 4, 5, 6, 7, 0]
final slot, lower 128 bits after: 0
```

Recommended Mitigation: Implement the following fix to load the existing value from storage and pack in the lower bits:

```
    sload(add(slot, mul(maxI, 32)))
```

Output after mitigation:

```
reservesBefore: [0, 0, 0, 0, 0, 0, 0, 420]
final slot, lower 128 bits before: 420
reserves: [1, 2, 3, 4, 5, 6, 7]
reserves2: [1, 2, 3, 4, 5, 6, 7, 420]
final slot, lower 128 bits after: 420
```

Beanstalk: Fixed in commit 5e61420.

Cyfrin: Acknowledged.

6.4 Informational

6.4.1 Non-standard storage packing

Per the Solidity docs, the first item in a packed storage slot is stored lower-order aligned; however, manual packing in `LibBytes` does not follow this convention. Modify the `storeUint128` function to store the first packed value at the lower-order aligned position.

Beanstalk: Fixed in commit 5e61420.

Cyfrin: Acknowledged.

6.4.2 EIP-1967 second pre-image best practice

When calculating custom EIP-1967 storage slots, as in Well.sol::RESERVES_STORAGE_SLOT, it is best practice to add an offset of `-1` to the hashed value to further reduce the possibility of a second pre-image attack.

Beanstalk: Fixed in commit dc0fe45.

Cyfrin: Acknowledged.

6.4.3 Remove experimental ABIEncoderV2 pragma

ABIEncoderV2 is enabled by default in Solidity 0.8, so two instances can be removed.

Beanstalk: All instances of ABIEncoderV2 pragma have been removed. Fixed in commit 3416a2d.

Cyfrin: Acknowledged.

6.4.4 Inconsistent use of decimal/hex notation in inline assembly

For readability and to prevent errors when working with inline assembly, decimal notation should be used for integer constants and hex notation for memory offsets.

Beanstalk: All new code written for Basin uses decimal notation. Decimal notation has been selected for all new code as it is more readable.

Some external libraries use hex notation (Ex. `ABDKMathQuad.sol`. It was decided that it is best to leave these libraries as is instead of modifying them to prevent complication.

Cyfrin: Acknowledged.

6.4.5 Unused imports and errors

In `LibMath`:

- OpenZeppelin SafeMath is imported but not used
- `PRBMath_MulDiv_Overflow` error is declared but never used

Beanstalk: Fixed here.

Cyfrin: Acknowledged.

6.4.6 Inconsistency in LibMath comments

There is inconsistent use of `x` in comments and `a` in code within the `nthRoot` and `sqrt` functions of `LibMath`.

Beanstalk: Fixed here.

Cyfrin: Acknowledged.

6.4.7 FIXME and TODO comments

There are several FIXME and TODO comments that should be addressed.

Beanstalk: Fixed here.

Cyfrin: Acknowledged.

6.4.8 Use correct NatSpec tags

Uses of `@dev See {IWell.fn}` should be replaced with `@inheritdoc IWell` to inherit the NatSpec documentation from the interface.

Beanstalk: Don't see any instances of `@dev See {IWell.fn}`. Removed similar instances of `See: {IAquifer.fn}`. Looking at the natspec documentation, it says "Functions without NatSpec will automatically inherit the documentation of their base function." Thus, it seems that adding @inheritdoc tags is unnecessary.

Cyfrin: Acknowledged.

6.4.9 Poorly descriptive variable and function names in `GeoEmaAndCumSmaPump` are difficult to read

For example, in `update`:

- `b` could be renamed `returnedReserves`.
- `aN` could be renamed `alphaN` or `alphaRaisedToTheDeltaTimeStamp`.

Additionally, `A`/`_A` could be renamed `ALPHA`, and `readN` could be renamed `readNumberOfReserves`.

Beanstalk: Fixed in commit 3bf0080.

- Rename `Reserves` struct to `PumpState`.
- Rename `b` to `pumpState`.
- Rename `readN` to `readNumberOfReserves`.
- Rename `n` to `numberOfReserves`.
- Rename `_A` to `_alpha`.
- Rename `A` to `ALPHA`.
- Rename `aN` to `alphaN`.

Cyfrin: Acknowledged.

6.4.10 Remove TODO Check if bytes shift is necessary

In `LibBytes16::readBytes16`, the following line has a `TODO`:

```
mstore(add(reserves, 64), shl(128, sload(slot))) // TODO: Check if byte shift is necessary
```

Since two reserve elements' worth of data is stored in a single slot, the left shift is indeed needed. The following test shows how these are different:

```
function testNeedLeftShift() public {
    uint256 reservesSize = 2;
    uint256 slotNumber = 12345;
    bytes32 slot = bytes32(slotNumber);

    bytes16[] memory leftShiftreserves = new bytes16[](reservesSize);
    bytes16[] memory noShiftreserves = new bytes16[](reservesSize);

    // store some data in the slot
    assembly {
        sstore(
            slot,
            0x0000000000000000000000000000007b0000000000000000000000000000011
        )
    }

    // left shift
    assembly {
        mstore(add(leftShiftreserves, 32), sload(slot))
        mstore(add(leftShiftreserves, 64), shl(128, sload(slot)))
    }

    // no shift
    assembly {
        mstore(add(noShiftreserves, 32), sload(slot))
        mstore(add(noShiftreserves, 64), sload(slot))
    }
    assert(noShiftreserves[1] != leftShiftreserves[1]);
}
```

Beanstalk: Fixed here.

Cyfrin: Acknowledged.

6.4.11 Use underscore prefix for internal functions

For functions such as `getSlotForAddress`, it is more readable to have this function be named `_getSlotForAddress` so readers know it is an internal function. A similarly opinionated recommendation is to use `s_` for storage variables and `i_` for immutable variables.

Beanstalk: Decided to add the _ prefix to internal functions, but not the `s_` or `i_` prefix. Fixed in commit 86c471f.

Cyfrin: Acknowledged.

6.4.12 Missing test coverage for a number of functions

Consider adding tests for `GeoEmaAndCumSmaPump::getSlotsOffset`, `GeoEmaAndCumSmaPump::getDeltaTimestamp`, and `_getImmutableArgsOffset` to increase test coverage and confidence that they are working as expected.

Beanstalk: Fixed here.

Cyfrin: Acknowledged.

6.4.13 Use `uint256` over `uint`

`uint` is an alias for `uint256` and is not recommended for use. The variable size should be clarified, as this can cause issues when encoding data with selectors if the alias is mistakenly used within the signature string.

Beanstalk: Fixed in commit 7ca7d64.

Cyfrin: Acknowledged.

6.4.14 Use constant variables in place of inline magic numbers

When using numbers, it should be made clear what the number represents by storing it as a constant variable.

For example, in `Well.sol`, the calldata location of the pumps is given by the following:

```
uint dataLoc = LOC_VARIABLE + numberOfTokens() * 32 + wellFunctionDataLength();
```

Without additional knowledge, it may be difficult to read and so it is recommended to assign variables such as:

```
uint256 constant ONE_WORD = 32;
uint256 constant PACKED_ADDRESS = 20;
...
uint dataLoc = LOC_VARIABLE + numberOfTokens() * ONE_WORD + wellFunctionDataLength();
```

The same recommendation can be applied to inline assembly blocks, which perform shifts such that numbers like `248` and `208` have some verbose meaning.

Additionally, when packing values in immutable data/storage, the code would benefit from a note explicitly stating where this is the case, e.g. pumps.

Beanstalk: Converted instances of `32` and `20` to `ONE_WORD` and `PACKED_ADDRESS`. Fixed here.

Cyfrin: Acknowledged.

6.4.15 Insufficient use of NatSpec and comments on complex code blocks

Many low-level functions, such as `WellDeployer::encodeAndBoreWell`, are missing NatSpec documentation. Additionally, many math-heavy contracts and libraries can only be easier to understand with NatSpec and supporting comments.

Beanstalk: Added natspec to `encodeAndBoreWell`. Fixed here.

Cyfrin: Acknowledged.

6.4.16 Precision loss on large values transformed between log2 scale and the normal scale

In `GeoEmaAndCumSmaPump.sol::_init`, the reserve values are transformed into log2 scale:

```
byteReserves[i] = reserves[i].fromUIntToLog2();
```

This transformation implies a precision loss, particularly for large `uint256` values, as demonstrated by the following test:

```
function testUIntMaxToLog2() public {
uint x = type(uint).max;
bytes16 y = ABDKMathQuad.fromUIntToLog2(x);
console.log(x);
console.logBytes16(y);
assertEq(ABDKMathQuad.fromUInt(x).log_2(), ABDKMathQuad.fromUIntToLog2(x));
uint x_recover = ABDKMathQuad.pow_2ToUInt(y);
console.log(ABDKMathQuad.toUInt(y));
console.log(x_recover);
}
```

Consider explicit limiting of the reserve values to avoid precision loss.

Beanstalk: This is expected. Compressing a uint256 into a bytes16 can't possibly not lose precision as it is compressing 256 bits into 128 bits. E.g. there is only 113-bit decimal precision on the log operation. See here.

Cyfrin: Acknowledged.

6.4.17 Emit events prior to external interactions

To strictly conform to the Checks Effects Interactions pattern, it is recommended to emit events before any external interactions. Implementing this pattern is generally advised to ensure correct migration through state reconstruction, which in this case, it should not be affected given that all instances in `Well.sol` are protected by the `nonReentrant` modifier, but it is still good practice.

Beanstalk: Decided not to implement due to the complexity of the change and its optionality.

Cyfrin: Acknowledged.

6.4.18 Time Weighted Average Price oracles are susceptible to manipulation

It should be noted that on-chain TWAP oracles are susceptible to manipulation. Using them to power critical parts of any on-chain protocol is potentially dangerous.

Beanstalk: This is our best attempt at a manipulation-resistant oracle. Manipulation will always be possible, but we believe that there is significant protection against Oracle manipulation in our implementation.

Cyfrin: Acknowledged.

6.5 Gas Optimization

6.5.1 Simplify modulo operations

In `LibBytes::storeUint128` and `LibBytes::readUint128`, `reserves.lenth % 2 == 1` and `i % 2 == 1` can be simplified to `reserves.length & 1 == 1` and `i & 1 == 1`.

Beanstalk: Fixed in commit 9db714a.

Cyfrin: Acknowledged.

6.5.2 Branchless optimization

The `sqrt` function in `MathLib` and related comment should be updated to reflect changes in Solmate's `FixedPointMathLib` which now includes the branchless optimization `z := sub(z, lt(div(x, z), z))`.

Beanstalk: Fixed in commit 46b24ac.

Cyfrin: Acknowledged.