Basin
Findings & Analysis Report

2023-10-05

Table of contents

- [M-05] The constant product invariant can be broken.
- [M-06] There is a large precision error in sqrt calculation of lp
- [M-07] `boreWell` can be frontrun/DoS-d
- Proof of Concept
- [M-08] Treating of BLOCK_TIME as permanent will cause serious economic flaws in the oracle when block times change
- [M-09] Aquifer is vulnerable to Metamorphic Contract Attack
- [M-10] Transferout exclusive `feeOnTransfer` tokens will run out of well
- [M-11] `addLiquidity` Sandwich Attack for unbalanced token deposits
- [M-12] Single hardcoded cap used for multiple tokens in a pump causing some assets to be more stale, while having no effects on other stable assets
- [M-13] Useless Modifier and Bad Assumptions in Constructor

- Low Risk and Non-Critical Issues

  - 1. No need to write `Aquifer.sol/wellImplementation()` getter function instead make the mapping `wellImplementations` as a public state variable.
  - 2. No need to introduce a new local variable in this function `Well.sol/tokens()` , directly return the tokens array.
  - 3. `Well.sol/_swapFrom()` function using the some lines of code same as written in another function `getSwapOut()` , which is increasing the codesize and deployment cost, instead call the function `getSwapOut()` inside the `_swapFrom()`
  - 4. `Well.sol/swapTo()` function using the some lines of code same as written in another function `getSwapIn()` , which is increasing the codesize and deployment cost, instead call the function `getSwapIn()` inside `swapTo()`
  - 5. Function `Well.sol/_getIJ()` will always revert for two equal token addresses due to an if-else condition.
  - 6. In `MultiFlowPump.sol/update()` use `slot.readLastReserves()` to get the value of `numberOfReserves` instead of using `reserves.length`

- 7. The non-zero reserve condition is used in `MultiFlowPump.sol/_init()` and `MultiFlowPump.sol/update()` . Since `_init()` function is called by only `upadate()` function, this will be waste to check this same validation two times
- 8. Can use `numberOfReserves` instead of `byteReserves.length` , this saves the calculation of `bytesReserves.length` in `MultiFlowPump.sol/_init()`
- 9. `MultiFlowPump.sol/_capReserve()` can be set as a `pure` function since this doesn't view any state of chain.
- 10. Minor typo in commenting in `MultiFlowPump.sol/_capReserve()`
- 11. Better to declare the return variable as `emaReserves` instead of `reserves` as it gives more information about the return value in `MultiFlowPump.sol/readLastInstantaneousReserves()`
- 12. updating `lastReserves[i]` by using `_capReserves()` is not consistent in functions `MultiFlowPump.sol/readInstantaneousReserves()` and `MultiFlowPump.sol/update()`
- 13. Better to declare the return variable as `cumulativeReserves` instead of `reserves` as it gives more information about the return value in `MultiFlowPump.sol/readLastCumulativeReserves()`
- 14. No need to introduce a new local variable in this function `MultiFlowPump.sol/readCumulativeReserves()` , directly return the tokens array.

- Gas Optimizations
    - G-01 Access mappings directly rather than using accessor functions
    - G-02 public functions not called by the contract should be declared external instead
    - G-03 Amounts should be checked for 0 before calling a transfer
    - G-04 With assembly, .call (bool success) transfer can be done gas-optimized
    - G-05 Use constants instead of type(uintx).max

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Basin smart contract system written in Solidity. The audit took place between July 3—July 10 2023.

Wardens

86 Wardens contributed reports to the Basin:

1. Trust
2. kutugu
3. oakcobalt
4. erebus
5. a3yip6
6. Cosine
7. Eeyore
8. qpzm
9. CRIMSON-RAT-REACH (0xtotem, imkapadia, cergyk,paspe, vangrim, devblixt, 0xChuck, vani, escrow, and VictoryGod)
10. ptsanev
11. LokiThe5th
12. peanuts
13. 0xSmartContract
14. pontifex
15. tonisives
16. Inspecktor
17. sces60107
18. Qeew
19. MohammedRizwan
20. Brenzee
21. K42
22. 0xprinc
23. SM3_SS
24. seth_lawson
25. bigtone

26. TheSavageTeddy
27. glcanvas
28. Rolezn
29. Raihan
30. JCN
31. lsaudit
32. 0xn006e7
33. josephdara
34. pfapostol
35. hunter_w3b
36. 0xAnah
37. mahdirostami
38. 33audits
39. codegpt
40. te_aut
41. alexzoid
42. Deekshith99
43. radev_sw
44. Kaysoft
45. QiuhaoLi
46. 0xWaitress
47. Topmark
48. 2997ms
49. LosPollosHermanos (LemonKurd, jc1, and scaraven)
50. max10afternoon
51. JGcarv
52. kaveyjoe
53. ginlee
54. zhaojie

55. 0xkazim
56. DanielWang888
57. ziyou-
58. 8olidity
59. 0x11singh99
60. SY_S
61. ElCid
62. SAAJ
63. DavidGiladi
64. MIQUINHO
65. Strausses
66. Udsen
67. Eurovickk
68. CyberPunks (Stryder, and andrewprasaath)
69. twcctop
70. John
71. 404Notfound
72. Jorgect
73. ravikiranweb3
74. fatherOfBlocks

This audit was judged by alcueca

Final report assembled by PaperParachute.

Summary

The C4 analysis yielded an aggregated total of 14 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 13 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 56 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 27 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Basin repository, and is composed of 10 smart contracts written in the Solidity programming language and includes 1145 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (1)

[H-01] Pumps are not updated in the shift() and sync() functions, allowing oracle manipulation

Submitted by Eeyore, *also found by* LokiThe5th, *Trust (*1, 2*),* pontifex, oakcobalt, *and* Brenzee

https://github.com/code-423n4/2023-07-basin/blob/9403cf973e95ef7219622dbbe2a08396af90b64c/src/Well.sol#L352-

L377
https://github.com/code-423n4/2023-07-basin/blob/9403cf973e95ef7219622dbbe2a08396af90b64c/src/Well.sol#L590-L598

The `Well` contract mandates that the `Pumps` should be updated with the previous block's `reserves` in case `reserves` are changed in the current block to reflect the price change accurately.

However, this doesn't happen in the `shift()` and `sync()` functions, providing an opportunity for any user to manipulate the `reserves` in the current block before updating the `Pumps` with new manipulated `reserves` values.

Impact

The `Pumps` (oracles) can be manipulated. This can affect any contract/protocol that utilizes `Pumps` as on-chain oracles.

Proof of Concept

1. A malicious user performs a `shift()` operation to update `reserve` s to desired amounts in the current block, thereby overriding the `reserves` from the previous block.
2. The user performs `swapFrom()/swapTo()` operations to extract back the funds used in the `shift()` function. As a result, the attacker is not affected by any arbitration as pool `reserves` revert back to the original state.
3. The `swapFrom()/swapTo()` operations trigger the `Pumps` update with invalid `reserves` , resulting in oracle manipulation.

Note: The `sync()` function can also manipulate `reserves` in the current block, but it's less useful than `shift()` from an attacker's perspective.

PoC Tests

This test illustrates how to use `shift()` to manipulate `Pumps` data.

Create `test/pumps/Pump.Manipulation.t.sol` and run `forge test --match-test manipulatePump`.

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

import {TestHelper, Call} from "../TestHelper.sol";
import {MultiFlowPump} from "src/pumps/MultiFlowPump.sol";
import {from18} from "test/pumps/PumpHelpers.sol";

contract PumpManipulationTest is TestHelper {
    MultiFlowPump pump;

    function setUp() public {
        pump = new MultiFlowPump(
            from18(0.5e18), // cap reserves if changed +/- 50% p
            from18(0.5e18), // cap reserves if changed +/- 50% p
            12, // block time
            from18(0.9e18) // ema alpha
        );

        Call[] memory _pumps = new Call[](1);
        _pumps[0].target = address(pump);
        _pumps[0].data = new bytes(0);

        setupWell(2,_pumps);
    }

    function test_manipulatePump() public prank(user) {
        uint256 amountIn = 1 * 1e18;

        // 1. equal swaps, reserves should be unchanged
        uint256 amountOut = well.swapFrom(tokens[0], tokens[1],
        well.swapFrom(tokens[1], tokens[0], amountOut, 0, user,

        uint256[] memory lastReserves = pump.readLastReserves(ac
        assertApproxEqAbs(lastReserves[0], 1000 * 1e18, 1);
        assertApproxEqAbs(lastReserves[1], 1000 * 1e18, 1);

        // 2. equal shift + swap, reserves should be unchanged
        increaseTime(120);

        tokens[0].transfer(address(well), amountIn);
        amountOut = well.shift(tokens[1], 0, user);
```

```
            well.swapFrom(tokens[1], tokens[0], amountOut, 0, user,

            lastReserves = pump.readLastReserves(address(well));
            assertApproxEqAbs(lastReserves[0], 1000 * 1e18, 1);
            assertApproxEqAbs(lastReserves[1], 1000 * 1e18, 1);
        }
    }
```

Tools Used

Foundry

Recommended Mitigation Steps

Update `Pumps` in the `shift()` and `sync()` function.

```
    function shift(
        IERC20 tokenOut,
        uint256 minAmountOut,
        address recipient
    ) external nonReentrant returns (uint256 amountOut) {
        IERC20[] memory _tokens = tokens();
-       uint256[] memory reserves = new uint256[](_tokens.length
+       uint256[] memory reserves = _updatePumps(_tokens.length)


    function sync() external nonReentrant {
        IERC20[] memory _tokens = tokens();
-       uint256[] memory reserves = new uint256[](_tokens.length
+       uint256[] memory reserves = _updatePumps(_tokens.length)
```

[publiuss (Basin) confirmed and commented](#):

> This issue has been fixed by updating the Pumps in `shift(...)` and `sync(...)`:

- https://github.com/BeanstalkFarms/Basin/blob/91233a22005986aa7c9f3b0c67393842cd8a8e4d/src/Well.sol#L380

- https://github.com/BeanstalkFarms/Basin/blob/91233a22005986aa7c9f3b0c67393842cd8a8e4d/src/Well.sol#L628

Medium Risk Findings (13)

[M-01] Memory corruption in getBytes32FromBytes() can likely lead to loss of funds

Submitted by [Trust]()

The `LibBytes` library is used to read and store `uint128` types compactly for Well functions. The function `getBytes32FromBytes()` will fetch a specific index as `bytes32`.

```
/**
 * @dev Read the `i`th 32-byte chunk from `data`.
 */
function getBytes32FromBytes(bytes memory data, uint256 i) inter
    uint256 index = i * 32;
    if (index > data.length) {
        _bytes = ZERO_BYTES;
    } else {
        assembly {
            _bytes := mload(add(add(data, index), 32))
        }
    }
}
```

If the index is out of bounds in the data structure, it returns `ZERO_BYTES = bytes32(0)`. The issue is that the OOB check is incorrect. If `index=data.length`, the request is also OOB. For example: `data.length=0` -> array is empty, `data[0]` is undefined. `data.length=32` -> array has one `bytes32`, `data[32]` is undefined.

In other words, fetching the last element in the array will return whatever is stored in memory after the `bytes` structure.

Impact

Users of `getBytes32FromBytes` will receive arbitrary incorrect data. If used to fetch reserves like `readUint128`, this could easily cause severe damage, like incorrect pricing, or wrong logic that leads to loss of user funds.

PoC

The damage is easily demonstrated using the example below:

```
pragma solidity 0.8.17;

contract Demo {
    event Here();
    bytes32 private constant ZERO_BYTES = bytes32(0);


    function corruption_POC(bytes memory data1, bytes memory dat
        _bytes = getBytes32FromBytes(data1, 1);
    }

    function getBytes32FromBytes(bytes memory data, uint256 i) i
        uint256 index = i * 32;
        if (index > data.length) {
            _bytes = ZERO_BYTES;
        } else {
            emit Here();
            assembly {
                _bytes := mload(add(add(data, index), 32))
            }
        }
    }
}
```

Calling corruption_POC with the following parameters:

```
{
        "bytes data1": "0x2222222222222222222222222222222222222
        "bytes data2": "0x3333333333333333333333333333333333333
}
```

The output `bytes32` is:

```
{
        "0": "bytes32: _bytes 0x0000000000000000000000000000000(
}
```

The 0x20 value is in fact the size of the `data2` bytes that resides in memory from the call to `getBytes32FromBytes`.

Recommended Mitigation Steps

Change the logic to:

```
if (index >= data.length) {
    _bytes = ZERO_BYTES;
} else {
    assembly {
        _bytes := mload(add(add(data, index), 32))
    }
}
```

publiuss (Basin) confirmed, but disagreed with severity and commented:

> The report is valid, but the function is not used in the code base and will be removed. Because of this, it was never tested and/or intended for use. Recommend medium.

alcueca (Judge) decreased severity to Medium and commented:

> The finding doesn't impact code in scope in a way that would lead to loss of funds. Instead, it would affect only future code.

publiuss (Basin) commented:

> The function has been removed from the codebase.

[M-02] Due to slot confusion, reserve amounts in the pump will be corrupted, resulting in wrong oracle values

Submitted by Trust, also found by kutugu

https://github.com/code-423n4/2023-07-basin/blob/c1b72d4e372a6246e0efbd57b47fb4cbb5d77062/src/libraries/LibBytes16.sol#L45
https://github.com/code-423n4/2023-07-basin/blob/c1b72d4e372a6246e0efbd57b47fb4cbb5d77062/src/libraries/LibLastReserveBytes.sol#L58

Description

The MultiFlowPump contract stores reserve counts on every update, using the libraries LibBytes16 and LibLastReserveBytes. Those libs pack `bytes16` values efficiently with the `storeBytes16()` and `storeLastReserves` functions. In case of an odd number of items, the last storage slot will be half full. Care must be taken to not step over the previous value in that slot. This is done correctly in `LibBytes`:

```
if (reserves.length & 1 == 1) {
    require(reserves[reserves.length - 1] <= type(uint128).max,
    iByte = maxI * 64;
    assembly {
        sstore(
            // @audit - overwrite SLOT+MAXI*32
            add(slot, mul(maxI, 32)),
                                                            // @audit
            add(mload(add(reserves, add(iByte, 32))), shl(128, s
        )
    }
}
```

As can be seen, it overwrites the slot with the previous 128 bits in the upper half of the slot, only setting the lower 128 bytes.

However, the wrong slot is read in the other two libraries. For example, in `storeLastReserves()`:

```
if (reserves.length & 1 == 1) {
    iByte = maxI * 64;
    assembly {
```

```
            sstore(
                // @audit - overwrite SLOT+MAXI*32
                add(slot, mul(maxI, 32)),
                                                                    // @audit
                add(mload(add(reserves, add(iByte, 32))), shr(128, s
            )
        }
    }
```

The error is not multiplying `maxI` before adding it to `slot`. This means that the reserves count encoded in lower 16 bytes in `add(slot, mul(maxI, 32))` will have the value of a reserve in a much lower index. Slots are used in 32 byte increments, i.e. S, S+32, S+64... When `maxI==0`, the intended slot and the actual slot overlap. When `maxI` is 1..31, the read slot happens to be zero (unused), so the first actual corruption occurs on `maxI==32`. By substitution, we get: `SLOT[32*32] = correct reserve | SLOT[32]` In other words, the 4rd reserve (stored in lower 128 bits of `SLOT[32]`) will be written to the 64th reserve.

The Basin pump is intended to support an arbitrary number of reserves safely, therefore the described storage corruption impact is in scope.

Impact

Reserve amounts in the pump will be corrupted, resulting in wrong oracle values.

PoC

1. A reserve update is triggered on the pump when some Well action occurs.
2. Suppose reserves are array `[0,1,2,...,63,64]`
3. Reserve count is odd, so affected code block is reached
4. `SLOT[32*32] = UPPER: 64 | LOWER: SLOT[32] = 64 | 3`

Recommended Mitigation Steps

Change the `sload()` operation in both affected functions to `sload(add(slot, mul(maxI, 32))`

publiuss (Basin) confirmed, but disagreed with severity and commented:

> This is a valid issue as the function incorrectly stores bytes, but it doesn't break anything of the Pump as the bytes that are incorrectly stored are never read.

> Regardless it should be fixed. Recommend changing to Medium.

alcueca (Judge) decreased severity to Medium and commented:

> The bug doesn't negatively impact the code in scope, only future code.

[M-03] Due to bit-shifting errors, reserve amounts in the pump will be corrupted, resulting in wrong oracle values

Submitted by Trust

It is advised to first read finding: `Due to slot confusion, reserve amounts in the pump will be corrupted, resulting in wrong oracle values`, which provides all the contextual information for this separate bug.

We've discussed how a wrong `sload()` source slot leads to corruption of the reserves. In `LibBytes16`, another confusion occurs.

Recall the correct storage overwriting done in `LibBytes`:

```
assembly {
    sstore(
        // @audit - overwrite SLOT+MAXI*32
        add(slot, mul(maxI, 32)),
                                                        // @audit - r
        add(mload(add(reserves, add(iByte, 32))), shl(128, shr(1
    )
}
```

Importantly, it **clears** the lower 128 bytes of the source slot and replaces the upper 128 bytes of the dest slot using the upper 128 bytes of the source slot:

`shl(128,shr(128,SOURCE))`

In `storeBytes16()` , the `shl()` operation has been discarded. This means the code will use the upper 128 bytes of SOURCE to overwrite the lower 128 bytes in DEST.

```
    if (reserves.length & 1 == 1) {
        iByte = maxI * 64;
        assembly {
            sstore(
            // @audit - overwrite SLOT+MAXI*32
                add(slot, mul(maxI, 32)),
                                                                        // @audit -
                add(mload(add(reserves, add(iByte, 32))), shr(128, s
            )
        }
    }
```

In other words, regardless of the SLOT being read, instead of keeping the lower 128 bits as is, it stores whatever happens to be in the upper 128 bits. Note this is a **completely** different error from the slot confusion, which happens to be in the same line of code.

Impact

Reserve amounts in the pump will be corrupted, resulting in wrong oracle values

PoC

Assume slot confusion bug has been corrected for clarity.

1. A reserve update is triggered on the pump when some Well action occurs.
2. Suppose reserves are array `[0,1,2,...,63,64]`
3. Suppose previous reserves are array `[P0,P1,...,P64]`
4. Reserve count is odd, so affected code block is reached
5. `SLOT[32*32] = UPPER: 64 | LOWER: UPPER(SLOT[32*32]) = 64 | P64`

Recommended Mitigation Steps

Replace the affected line with the calculation below: `shr(128, shl(128, SLOT))` This will use the lower 128 bytes and clear the upper 128 bytes,as intended.

publiuss (Basin) confirmed, but disagreed with severity and commented:

> This is a valid issue as the function incorrectly stores bytes, but it doesn't break anything of the Pump as the bytes that are incorrectly shifted are never non-zero.

> Regardless it should be fixed. Recommend changing to Medium.

alcueca (Judge) decreased severity to Medium and commented:

> @publiuss, I noticed that in your fix you didn't change any tests. May I suggest that you increase your test coverage to be certain that the wardens are not missing other storage corruption issues?

> Other than that, without a clear PoC, I can't accept this as High.

publiuss (Basin) commented:

> @publiuss, I notice that in your fix you didn't change any tests. May I suggest that you increase your test coverage to be certain that the wardens are not missing other storage corruption issues?

> Other than that, without a clear PoC, I can't accept this as High.

> The test coverage didn't change because the issue doesn't actually impact the functionality. This issue has been addressed and fixed in all bytes libraries.

[M-04] Long term denial of service due to lack of fees in Well

Submitted by Trust, also found by ptsanev

The Well allows users to permissionless swap assets or add and remove liquidity. Users specify the intended slippage in `swapFrom`, in `minAmountOut`.

The ConstantProduct2 implementation ensures `Kend - Kstart >= 0`, where `K = Reserve1 * Reserve2`, and the delta should only be due to tiny precision errors.

Furthermore, the Well does not impose any fees to its users. This means that all conditions hold for a successful DOS of any swap transactions.

1. Token cost of sandwiching swaps is zero (no fees) - only gas cost
2. Price updates are instantenous through the billion dollar formula.
3. Swap transactions along with the max slippage can be viewed in the mempool

Note that such DOS attacks have serious adverse effects both on the protocol and the users. Protocol will use users due to disfunctional interactions. On the other side, users may opt to increment the max slippage in order for the TX to go through, which can be directly abused by the same MEV bots that could be performing the DOS.

Impact

All swaps can be reverted at very little cost.

PoC

1. Evil bot sees swap TX, slippage=S
2. Bot submits a flashbot bundle, with the following TXs

    1. Swap TX in the same direction as victim, to bump slippage above S
    2. Victim TX, which will revert
    3. Swap TX in the opposite direction and velocity to TX (1). Because of the constant product formula, all tokens will be restored to the attacker.

Recommended Mitigation Steps

Fees solve the problem described by making it too costly for attackers to DOS swaps. If DOS does takes place, liquidity providers are profiting a high APY to offset the inconvenience caused, and attract greater liquidity.

publiuss (Basin) disputed and commented:

> This is an issue that is already present in other AMMs. The lack of a fee just makes the DOS cheaper than in other AMMs. However, it still requires paying for 2 Ethereum transaction fees. The use of a private mempool or a higher priority fee solves this problem.

[alcueca (Judge) decreased severity to Medium and commented](https://github.com/code-423n4/2023-07-basin-findings/issues/136#issuecomment-1685917064):

> It seems to me that the DoS can be economical enough for the attacker to disrupt the UX by forcing all users to use private mempools.

[M-05] The constant product invariant can be broken.

Submitted by [qpzm](), *also found by* [CRIMSON-RAT-REACH]()

[https://github.com/code-423n4/2023-07-basin/blob/main/src/functions/ConstantProduct2.sol#L65-L66]()
[https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L590-L598]()
[https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L695-L702]()
[https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L541]()
[https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L562]()
[https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L582]()

Description

Let `reserves` returned by `Well._getReserves()` as x, y and `Well.tokenSupply()` as k. They must maintain the invariant `x * y * EXP_PRECISION = k ** 2`. However, the reserves can increase without updating the token supply if a user transfers one token of the well and call `Well.sync()`. We can sync the reserves and balances using `Well.sync`, but there is no way to sync `Well.tokenSupply() ** 2` to `x * y * EXP_PRECISION`.

Impact

`ConstantProduct2.calcReserve` assumes `Well.tokenSupply` equals to `reserves[0] * reserves[1]`. This exception breaks the assumption and reverts normal transactions. For example, when `Well.totalSupply` is less than `reserves[0] * reserves[1]`, [Well.removeLiquidityImbalanced](https://github.com/code-423n4/2023-07-basin) may revert.

1. Comment out minting initial liquidity in `TestHelper.sol`. https://github.com/code-423n4/2023-07-basin/blob/main/test/TestHelper.sol#L107

```
function setupWell(Call memory _wellFunction, Call[] memory _pun
    // ...
    // @audit comment out the line 107 to see apparently
    // Add initial liquidity from TestHelper
    // addLiquidityEqualAmount(address(this), initialLiquidity);
}
```

2. Add the test in `Well.AddLiquidity.t.sol` as below. https://github.com/code-423n4/2023-07-basin/blob/main/test/Well.AddLiquidity.t.sol#L9

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

    // @audit add this test
    function test_tokenSupplyError() public {
        IERC20[] memory tokens = well.tokens();
        Balances memory userBalance;
        Balances memory wellBalance = getBalances(address(well),

        console.log(wellBalance.lpSupply); // 0

        mintTokens(user, 10000000e18);

        vm.startPrank(user);
        tokens[0].transfer(address(well), 100);
        tokens[1].transfer(address(well), 100);
        vm.stopPrank();

        userBalance = getBalances(user, well);
        console.log(userBalance.lp); // 0

        addLiquidityEqualAmount(user, 1);
```

```
        userBalance = getBalances(user, well);
        console.log(userBalance.lp); // 1e6

        well.sync(); // reserves = [101, 101]

        uint256[] memory amounts = new uint256[](tokens.length);
        amounts[0] = 1;

        // FAIL: Arithmetic over/underflow
        vm.prank(user);
        well.removeLiquidityImbalanced(type(uint256).max, amount
    }
}
```

3. I commented the reason of underflow. https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L562

```
function removeLiquidityImbalanced(
    uint256 maxLpAmountIn,
    uint256[] calldata tokenAmountsOut,
    address recipient,
    uint256 deadline
) external nonReentrant expire(deadline) returns (uint256 lpAmou
    IERC20[] memory _tokens = tokens();
    uint256[] memory reserves = _updatePumps(_tokens.length);

    for (uint256 i; i < _tokens.length; ++i) {
        _tokens[i].safeTransfer(recipient, tokenAmountsOut[i]);
        reserves[i] = reserves[i] - tokenAmountsOut[i];
    }

    // @audit
    // 1e6 - sqrt(101 * (101 - 1) * 1000000 ** 2)
    // <=> 1e6 - 100498756
    lpAmountIn = totalSupply() - _calcLpTokenSupply(wellFunctior
    if (lpAmountIn > maxLpAmountIn) {
        revert SlippageIn(lpAmountIn, maxLpAmountIn);
    }
    _burn(msg.sender, lpAmountIn);

    _setReserves(_tokens, reserves);
    emit RemoveLiquidity(lpAmountIn, tokenAmountsOut, recipient)
```

```
    }
```

Recommended Mitigation Steps

In `Well.sync()`, mint `(reserves[0] * reserves[1] * ConstantProduct2.EXP_PRECISION).sqrt() - totalSupply()` Well tokens to `msg.sender`.

This keeps the invariant that `Well.tokenSupply() ** 2` equals to `reserves[0] * reserves[1] * ConstantProduct2.EXP_PRECISION` as long as the swap fee is 0.

[publiuss (Basin) confirmed via duplicate issue #210](#)

[trust1995 (Warden) commented](#):

> This is a good find. However it is hard to find rationalization for HIGH impact. Root issue is: attacker can make it so that there is more funds in the pool than there are supposed to be. The loser of the donation + sync() transaction is the attacker! LP providers will be able to redeem their shares with either:

1. `removeLiquidityOneToken()`
2. `removeLiquidity()`

> Specifically, `removeLiquidityImbalanced` will revert because of underflow, but there is no loss or freeze of funds. Therefore it seems clear that maximum severity would be Medium, because a one specific functionality is blocked. For High, according to the C4 severity guidelines warden must demonstrate a viable loss of funds or breaking the core of the protocol. That is not the case here.

[alcueca (Judge) commented](#):

> After careful consideration, I'm going to disagree with @trust1995, here is why.

> The nature of a DoS is that it is temporary. Either because it takes resources from the attacker to maintain the DoS, or because the victim can eventually change its configuration to stop the DoS.

> In this case, the DoS is permanent. The attacker can disable a certain functionality permanently and in all pools. The only possible remedy for Moonwell would be to convince all LPs to remove liquidity, and then to add it again in fixed Wells, which is completely unrealistic.

> To me, losing forever a feature is worse than not being able to operate at all for a short period of time. You don't ever fully recover. Moreover, the invariant of the pools is also broken forever.

> There are no severities between Medium and High, so if I have to choose, I'll have to choose High for this one.

trust1995 (Warden) commented:

> I appreciate the thought you have put into the submission.

> However, I believe "losing a feature, forever" is not a HIGH severity rationalization, and there is no case law in C4 to support that it is. I refer you to the two leading severity standards, the **C4 standard** and the **Immunefi standard**. Here is how C4 differentiates between Med and High:

> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements. 3 — High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).

> Clearly this is a "function of the protocol or its availability could be impacted" scenario.

> Here's how Immunefi would classify the issue:

> Low - Contract fails to deliver promised returns, but doesn't lose value: This is when the code doesn't work as intended (i.e. there is some logic error but that logic error doesn't affect the protocol's funds or user funds). Another example would be an external function that is meant to return a value does not return the correct thing, however, this value is not used elsewhere in application logic.

> The last thing I would say is the following plot twist - we are actually NOT even losing a functionality forever. There is an easy bypass to calling

> `removeLiquidityImbalanced()` . Instead, simply call the two functions:

1. removeLiquidity() - remove any balanced amount of liquidity
2. removeLiquidityOneToken() - remove any remaining liquidity of the higher between the token amounts.

> In a worst case scenario, the frontend can implement imbalanced withdrawals with these two calls or just the 2nd call.

> Most judges would consider it QA as there is basically no impact, inconvenience at most. To call it a HIGH would be, from my professional opinion, unthinkable.

[lokithe5th (Warden) commented]:

> Please forgive me if this is inappropriate @alcueca and @trust1995 , but I might add the following evidence for consideration:

> Fixing the imbalance is trivial, and will likely happen by accident if another user adds liquidity. This is because of the same mechanism [as described in this PoC].

> This boils down to a quirk of this specific implementation: when there is a discrepancy between the reserves and the `totalSupply` , it is corrected in the next call to `addLiquidity` or `swapFrom` .

> However, it must be noted that this may open up the contract to more consistent DoS attacks: the attacker can donate and force underflow, and then regain their attack funds by calling `addLiquidity` or `swapFrom` , but at the cost of removing the block. This would be a risky attack, as any user can front-run and steal the attacker's donation.

[alcueca (Judge) decreased severity to Medium and commented]:

> Thanks @lokithe5th, it is true that the DoS is not permanent and easily reversed. Downgraded to Medium.

[publiuss (Basin) commented]:

> This was addressed by modifying the `sync()` function to to have the signature `sync(address recipient, uint256 minLpAmountOut)` and mint LP tokens to

> `recipient` address to prevent the invariant from breaking. It now behaves like a `shift(...)` for adding liquidity instead of swapping.

[M-06] There is a large precision error in sqrt calculation of lp

Submitted by [kutugu]

Compared with div, there is a larger precision error in calculating lp through sqrt, so there should be a way to check whether there are excess tokens left when adding liquidity.

Proof of Concept

```
    function testCalcLpTokenSupplyDiff() public {
        uint256[] memory reserves = new uint256[](2);
        reserves[0] = 1e24 + 1e4;
        reserves[1] = 10000;
        uint256 lp1 = this.calcLpTokenSupply(reserves, bytes(""));
        reserves[0] = 1e24;
        reserves[1] = 10000;
        uint256 lp2 = this.calcLpTokenSupply(reserves, bytes(""));
        assert(lp1 == lp2);
    }
```

When reserve[0] is larger relative to reserve[1], the accuracy error is larger, and unlike div, the accuracy error is only 1, the accuracy error of sqrt is larger. When the user input the imbalance the amount will left excess reserve, searchers will monitor the contract after the excess reserve accumulation to a certain degree, will withdraw them by removeLiquidityImbalanced.

Recommended Mitigation Steps

Excess reserve tokens should be returned when the user adds liquidity

publiuss (Basin) acknowledged and commented:

> This is a known issue. The documentation should be updated.

> This was appended to the documentation here.

[M-07] `boreWell` can be frontrun/DoS-d

Submitted by tonisives, also found by Inspecktor, peanuts, sces60107, Qeew, and MohammedRizwan

The `boreWell` function in the Aquifer contract is responsible for creating new Wells. However, there are two critical security issues:

1. **Stealing of user's deposit amount**: The public readability of the `salt` parameter allows an attacker to frontrun a user's transaction and capture the deposit amount intended for the user's Well. By creating a Well with the same `salt` value, the attacker can receive the deposit intended for the user's Well and withdraw the funds.
2. **DoS for `boreWell`**: Another attack vector involves an attacker deploying a Well with the same `salt` value as the user's intended Well. This causes the user's transaction to be reverted, resulting in a denial-of-service (DoS) attack on the `boreWell` function. The attacker can repeatedly execute this attack, preventing users from creating new Wells.

Proof of Concept

Stealing of user's deposit amount

If a user intends to create a new Well and deposit funds into it, an attacker can frontrun the user's transactions and capture the deposit amount. Here is how the attack scenario unfolds:

1. The user broadcasts two transactions: the first to create a Well with a specific `salt` value, and the second to deposit funds into the newly created Well.
2. The attacker views these pending transactions and frontruns them by creating a Well for themselves using the same `salt` value.
3. The attacker's Well gets created with the same address that the user was expecting for their Well.

4. As a result, the user's create Well transaction gets reverted, but the deposit transaction successfully executes, depositing the funds into the attacker's Well.
5. Being the owner of the Well, the attacker can simply withdraw the deposited funds from the Well.

DoS for `boreWell`

In this attack scenario, an attacker can forcefully revert a user's create Well transaction by deploying a Well for themselves using the user's `salt` value. Here are the steps of the attack:

1. The user broadcasts a create Well transaction with a specific `salt` value.
2. The attacker frontruns the user's transaction and creates a Well for themselves using the same `salt` value.
3. As a result, the user's original create Well transaction gets reverted since the attacker's Well already exists at the predetermined address.
4. This attack can be repeated multiple times, effectively causing a denial-of-service (DoS) attack on the boreWell function.

Tools Used

VS Code

Recommended Mitigation Steps

To mitigate the identified security issues, it is recommended to make the upcoming Well address user-specific by combining the `salt` value with the user's address. This ensures that each user's Well has a unique address and prevents frontrunning attacks and DoS attacks. The following code snippet demonstrates the recommended modification:

```
well = implementation.cloneDeterministic(
    keccak256(abi.encode(msg.sender, `salt`))
);
```

[publiuss (Basin) confirmed and commented](https://github.com/code-423n4/2023-07-basin-findings/issues/):

> This issue has been addressed in the code. The `boreWell(...)` function now uses a `salt` consisting of the hash of `msg.sender` appended to the input `salt` value.

> See here.

[M-08] Treating of BLOCK_TIME as permanent will cause serious economic flaws in the oracle when block times change

Submitted by Trust

Pumps receive the chain BLOCK*TIME in the constructor. In every update, it is used to calculate the* `blocksPassed` *variable, which determines what is the maximum change in price (done in* `capReserve()`).

The issue is that BLOCK*TIME is an immutable variable in the pump, which is immutable in the Well, meaning it is basically set in stone and can only be changed through a Well redeploy and liquidity migration (very long cycle). However, BLOCK*TIME actually changes every now and then, especially in L2s.For example, the recent Bedrock upgrade in Optimism completely changed the block time generation. It is very clear this will happen many times over the course of Basin's lifetime.

When a wrong BLOCK*TIME is used, the* `capReserve()` function will either limit price changes too strictly, or too permissively. In the too strict case, this would cause larger and large deviations between the oracle pricing and the real market prices, leading to large arb opportunities. In the too permissive case, the function will not cap changes like it is meant to, making the oracle more manipulatable than the economic model used when deploying the pump.

Impact

Treating of BLOCK_TIME as permanent will cause serious economic flaws in the oracle when block times change.

Recommended Mitigation Steps

The BLOCK_TIME should be changeable, given a long enough freeze period where LPs can withdraw their tokens if they are unsatisfied with the change.

[publiuss (Basin) confirmed and commented]:

> This issue was addressed by (1) changing the variable name from `BLOCK_TIME` to `CAP_INTERVAL` and (2) rounding up when calculating `capInterval` (See [here].)

> (1) By changing the name it is clear that this parameter does not have to be the block time. (2) By rounding up, the system protects itself against the case where the chain block chain is greater than `CAP_INTERVAL` capExponent will always be non-zero when time has passed.

[M-09] Aquifer is vulnerable to Metamorphic Contract Attack

Submitted by [a3yip6]

The [Aquifer] contract supports multiple ways to deploy the `Well` contracts. More specifically, it supports `create` and `create2` at the same time. However, such a feature is vulnerable to the [Metamorphic Contract Attack]. That is to say, attackers are capable to deploy two different `Well` implementations in the same address, which is recorded by `mapping(address => address) wellImplementations;`.

Although the Aquifer contract is claimed to be permissionless, it should not break the immutability. Thus, we consider it a medium-risk bug.

Impact

The real implementation of the `Well` contract listed in `Aquifer` may be inconsistent with the expectation of users. Even worse, users may suffer from unexpected loss due to the change of contract logic.

Proof of Concept

```
    // the Aquifer contract
    function boreWell(
        address implementation,
```

```
    bytes calldata immutableData,
    bytes calldata initFunctionCall,
    bytes32 salt
) external nonReentrant returns (address well) {
    if (immutableData.length > 0) {
        if (salt != bytes32(0)) {
            well = implementation.cloneDeterministic(immutableDa
        } else {
            well = implementation.clone(immutableData);
        }
    } else {
        if (salt != bytes32(0)) {
            well = implementation.cloneDeterministic(salt);
        } else {
            well = implementation.clone();
        }
    }
    ...
}

// the cloneDeterministic() function
function cloneDeterministic(address implementation, bytes32 salt
        internal
        returns (address instance)
    {
        /// @solidity memory-safe-assembly
        assembly {
            mstore(0x21, 0x5af43d3d93803e602a57fd5bf3)
            mstore(0x14, implementation)
            mstore(0x00, 0x602c3d8160093d39f33d3d3d3d363d3d37363
            instance := create2(0, 0x0c, 0x35, salt)
            // Restore the part of the free memory pointer that
            mstore(0x21, 0)
            // If `instance` is zero, revert.
            if iszero(instance) {
                // Store the function selector of `DeploymentFai
                mstore(0x00, 0x30116425)
                // Revert with (offset, size).
                revert(0x1c, 0x04)
            }
        }
    }
```

As shown in the above code, attackers are capable to deploy new `Well` contracts through `cloneDeterministic` multiple times with the same input parameter `implementation`. And the `cloneDeterministic` function utilizes the following bytecode to deploy a new `Well` contract: `0x602c3d8160093d39f33d3d3d3d363d3d37363d73 + implementation + 5af43d3d93803e602a57fd5bf3`. That is to say, if the address (i.e., `implementation`) remains the same, then the address of the deployed `Well` contract also remains the same.

Normally, EVM would revert if anyone re-deploy a contract to the same address. However, if the `implementation` contract contains self-destruct logic, then attackers can re-deploy a new contract with different bytecode to the same address through `cloneDeterministic`.

Here is how we attack:

- Assuming Bob deploys `Well_Implementation1` to address 1.
- Bob invoke `Aquifer:boreWell` with address 1 as the parameter to get a newly deployed `Well1` contract at address 2.
- Bob invokes the self-destruct logic in the `Well_Implementation1` contract and re-deploy a new contract to address 1 through [Metamorphic Contract](), namely `Well_Implementation2`.
- Bob invoke `Aquifer:boreWell` with address 1 again. Since the input of `create2` remains the same, a new contract is deployed to address 2 with new logic from `Well_Implementation2`.

Recommended Mitigation Steps

Remove the `cloneDeterministic` feature, leaving the `clone` functionality only.

[publiuss (Basin) acknowledged and commented]():

> This issue is only an issue if an implementation address contains a way to self-destruct itself. No implementation address should be considered valid if it contains a way to self-destruct. This should be probably documented in all documentation.

[alcueca (Judge) commented]():

> Agree that this should be documented. It is pertinent to those auditing Wells.

[publiuss (Basin) commented]():

> It was documented [here]().

> That Well implementations should not be able to self-destruct.

[M-10] Transferout exclusive `feeOnTransfer` tokens will run out of well

Submitted by [kutugu]()

https://github.com/code-423n4/2023-07-basin/blob/9403cf973e95ef7219622dbbe2a08396af90b64c/src/Well.sol#L610
https://github.com/code-423n4/2023-07-basin/blob/9403cf973e95ef7219622dbbe2a08396af90b64c/src/Well.sol#L304
https://github.com/code-423n4/2023-07-basin/blob/9403cf973e95ef7219622dbbe2a08396af90b64c/src/Well.sol#L370

Impact

The well does not check the actual transferout amount and the k value, which for exclusive `feeOnTransfer` tokens causes the well to have a portion of the token fee cut out of air every time the token is transferred, which is not included in the transfer amount. Attackers can take advantage of this to run out of well.

Proof of Concept

The most common attack flow is through the skim function:

1. Attacker swaps to increase the price of exclusive `feeOnTransfer` token
2. Attacker transfers token to well and skim. This will cut some of the well funds
3. Repeat the above process to reduce the number of well tokens to 1
4. Attacker calls sync and swap to pair token

Recommended Mitigation Steps

Check the correct amount every time you transfer out.

alcueca (Judge) commented:

> I think this is correct. Most fee-on-transfer tokens will take the fee on the recipient, but there isn't a rule or even a standard that forbids the fee-on-transfer token to take the fees from the sender, or from both. My bank has all of these options for transfers that include fees.

> The Well in scope is intended to support fee-on-transfer tokens, but nowhere it is specified that it is intended to support only recipient-fee-on-transfer tokens, and therefore is vulnerable. This could be a critical except that recipient-fee-on-transfer tokens are exceedingly rare at the time, and only Wells deployed for those exceedingly rare tokens would be drained.

publiuss (Basin) disputed and commented:

> `Skim` will only transfer tokens out of the Well if the balances of tokens in the well are greater than the reserves of the Well. This only happens if someone sends tokens to the Well without calling `sync` or `shift`.

> There is no way to abuse this mechanism as the Well's token balances can never drop below reserve balances as a result of skim.

alcueca (Judge) commented:

> @publiuss, I can reduce the severity to QA on account of the low quality and invalid Proof of Concept, but I think that the attack vector is valid.

> Correct me if I'm wrong, but swapFromFeeOnTransfer only considers fees in the incoming token. For the outgoing token it assumes that fees will be deducted from the amount received, and calls _swapFrom as for non-fee tokens.

> Normally I wouldn't worry too much about fee-on-transfer tokens, but you explicitly intend to support them, and there is no standard defining how fees should be collected, so they could be deducted from the sender's balance after each successful transfer.

> During a swapFromFeeOnTransfer, inside the _swapFrom call, we will calculate what the reserves should be, and from there the amountOut. Then the Well transfers out the amountOut and sets the reserves to the calculated amounts.

> Only that if a fee is collected from the sender after the transfer, the stored reserves will be higher than the actual well balances, and this discrepancy will grow with each swap.

> Would you please check again if my reasoning is right?

> I don't know if I would fix this in the code, I would most likely just state in the docs that only fee-on-transfer tokens where the fee is collected from the receiver are supported, and those where the fee is collected from the sender are not, but that is up to you.

publiuss (Basin) commented:

> Hello @alcueca, your reasoning is correct and I agree with your thoughts in regards to the solution.

[M-11] `addLiquidity` Sandwich Attack for unbalanced token deposits

Submitted by Cosine

https://github.com/code-423n4/2023-07-basin/blob/c1b72d4e372a6246e0efbd57b47fb4cbb5d77062/src/Well.sol#L392-L399

https://github.com/code-423n4/2023-07-basin/blob/c1b72d4e372a6246e0efbd57b47fb4cbb5d77062/src/Well.sol#L495-L517

Impact

Wells supports adding and removing liquidity in imbalanced proportions. If a user wants to deposit liquidity in an imbalanced ratio (only one token). A attacker can front run the user by doing the same and removing the liquidity directly after the deposit of the user. By doing so the attacker steals a significant percentage of the users funds. This bug points to a mathematical issue whose effects could be even greater.

Proof of Concept

The following code demonstrates the described vulnerability:

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

import {TestHelper, Balances} from "test/TestHelper.sol";

contract Exploits is TestHelper {
    function setUp() public {
        setupWell(2); // Setup Well with two tokens and a balanc
    }

    function test_add_liquidity_sandwich_attack() public {
        // init attacker and liquidityProvider, gives them some
        address liquidityProvider = vm.addr(1);
        address attacker = vm.addr(2);
        mintTokens(liquidityProvider, uint256(100 * 1e18));
        mintTokens(attacker, uint256(1000 * 1e18));
        Balances memory liquidityProviderBalanceBefore = getBala
        Balances memory attackerBalanceBefore = getBalances(addr

        // liquidityProvider wants to provide liquidity for only

        // attacker front runs liquidityProvider and add liquidi
        vm.startPrank(attacker);
        tokens[0].approve(address(well), 1000 * 1e18);
        uint256[] memory amounts1 = new uint256[](2);
        amounts1[0] = 1000 * 1e18; // 10x the tokens of the liqu
        amounts1[1] = 0;
        well.addLiquidity(amounts1, 0, attacker, type(uint256).m
        vm.stopPrank();

        // liquidityProvider deposits some token[0] tokens
        vm.startPrank(liquidityProvider);
        tokens[0].approve(address(well), 100 * 1e18);
        uint256[] memory amounts2 = new uint256[](2);
        amounts2[0] = 100 * 1e18;
        amounts2[1] = 0;
```

```
        well.addLiquidity(amounts2, 0, liquidityProvider, type(u
        vm.stopPrank();

        // attacker burns the LP tokens directly after the depos
        vm.startPrank(attacker);
        Balances memory attackerBalanceBetween = getBalances(add
        well.removeLiquidityOneToken(
            attackerBalanceBetween.lp,
            tokens[0],
            1,
            address(attacker),
            type(uint256).max
        );
        vm.stopPrank();
        Balances memory attackerBalanceAfter = getBalances(addre

        // the attacker got nearly 30% of the liquidityProviders
        // the percentage value can be increased even further by
        // and/or decreasing the amount of tokens the liquidityP
        assertTrue(attackerBalanceAfter.tokens[0] - attackerBala
    }
}
```

The test above can be implemented in the Basin test suite and is executable with the following command:

```
forge test --match-test test_add_liquidity_sandwich_attack
```

Tools Used

Foundry, VSCode

Recommended Mitigation Steps

Investigating the math behind this function further and writing tests for this edge case is necessary. A potential fix could be forcing users to provide liquidity in the current ratio of the reserves.

publiuss (Basin) disputed and commented:

> This can be mitigated through the use of the `minLpAmountOut` parameter in the function (If the attacker changes the ratio of reserves in the pool, then the number of LP tokens the user receives will be different). The POC provided uses a `minLpAmountOut` of 0 and thus the manipulation is succesful.

> Note: This behaves the same as Curve pools.

alcueca (Judge) decreased severity to Medium and commented:

> Front-running liquidity operations is possible in other live AMMs, only made more efficient by the lack of fees. Using user-defined slippage controls is an adequate mitigation.

[M-12] Single hardcoded cap used for multiple tokens in a pump causing some assets to be more stale, while having no effects on other stable assets

Submitted by oakcobalt

https://github.com/code-423n4/2023-07-basin/blob/c1b72d4e372a6246e0efbd57b47fb4cbb5d77062/src/pumps/MultiFlowPump.sol#L36-L37

https://github.com/code-423n4/2023-07-basin/blob/c1b72d4e372a6246e0efbd57b47fb4cbb5d77062/src/pumps/MultiFlowPump.sol#L205-L208

https://github.com/code-423n4/2023-07-basin/blob/c1b72d4e372a6246e0efbd57b47fb4cbb5d77062/src/pumps/MultiFlowPump.sol#L212-L215

Impact

When multiple tokens are handled in MultiFlowPump.sol, during trading, the pump might either unfairly reflects volatile assets to be more stale than normal trading activities, Or may fail to have any effects on relatively more stable assets.

This could distort SMA and EMA token reserve values and unfairly reflect some token prices because it uses a one-size-fits-all approach to multiple token asset prices indiscriminate of their normal volatility levels.

Proof of Concept

In MultiFlowPump.sol, a universal cap of maximum percentage change of a token reserve is set up as immutable variables - `LOG_MAX_INCREASE` and `LOG_MAX_DECREASE`. These two values are benchmarks for all tokens handled by the pump and are used to cap token reserve change per block in `_capReserve()`. And this cap is applied and checked on every single `update()` invoked by Well.sol.

```
//MultiFlowPump.sol
    bytes16 immutable LOG_MAX_INCREASE;
    bytes16 immutable LOG_MAX_DECREASE;
```

```
//MultiFlowPump.sol-update()
...
  for (uint256 i; i < numberOfReserves; ++i) {
            // Use a minimum of 1 for reserve. Geometric means w
|>            pumpState.lastReserves[i] = _capReserve(
                pumpState.lastReserves[i], (reserves[i] > 0 ? re
            );
...
```

```
//MultiFlowPump.sol-_capReserve()
...
        if (lastReserve.cmp(reserve) == 1) {
|>            bytes16 minReserve = lastReserve.add(blocksPassed.
            // if reserve < minimum reserve, set reserve to mini
            if (minReserve.cmp(reserve) == 1) reserve = minReser
        }
        // Rerserve Increasing or staying the same.
        else {
|>            bytes16 maxReserve = blocksPassed.mul(LOG_MAX_INCF
            maxReserve = lastReserve.add(maxReserve);
            // If reserve > maximum reserve, set reserve to maxi
            if (reserve.cmp(maxReserve) == 1) reserve = maxReser
        }
        cappedReserve = reserve;
```

```
...
```

As seen from above, whenever a token reserve change is over the percentage dictated by `LOG_MAX_INCREASE` or `LOG_MAX_DECREASE` , the token reserve change will be capped at the maximum percentage and rewritten as calculated `minReserve` or `maxReserve` value. This is fine if the pump is only managing one trading pair(two tokens) because their trading volatility level can be determined.

However, this is highly vulnerable when multiple trading pairs and multiple token assets are handled by a pump. And this is the intended scenario, which can be seen in Well.sol `_updatePumps()` where all token reserves handled by well will be passed to MultiFlowPump.sol. In this case, either volatile tokens will become more stale compared to their normal trading activities, or some stable tokens are allowed to deviate more than their normal trading activities which are vulnerable to exploits.

See POC below as an example to show the above scenario. Full test file here.

```
//Pump.HardCap.t.sol
...
 function setUp() public {
        mWell = new MockReserveWell();
        initUser();
        pump = new MultiFlowPump(
            from18(0.5e18),
            from18(0.333333333333333333e18),
            12,
            from18(0.9e18)
        );
    }

    function test_HardCap() public {
        uint256[] memory initReserves = new uint256[](4);
        //initiate for mWell and pump
        initReserves[0] = 100e8; //wBTC
        initReserves[1] = 1600e18; //wETH
        initReserves[2] = 99000e4; //USDC
        initReserves[3] = 98000e4; //USDT
        mWell.update(address(pump), initReserves, new bytes(0));
        increaseTime(12);
        //Reserve update
        uint256[] memory updateReserves = new uint256[](4);
```

```
        updateReserves[0] = 160e8; //wBTC
        updateReserves[1] = 1000e18; //wETH
        updateReserves[2] = 96000e4; //USDC
        updateReserves[3] = 101000e4; //USDT
        mWell.update(address(pump), updateReserves, new bytes(0)
        // lastReserves0 reflects initReserves[i]
        uint256[] memory lastReserves0 = pump.readLastReserves(a
        increaseTime(12);
        mWell.update(address(pump), updateReserves, new bytes(0)
        // lastReserves1 reflects 1st reserve update
        uint256[] memory lastReserves1 = pump.readLastReserves(a
        assertEq(
            ((lastReserves1[0] - lastReserves0[0]) * 1000) / las
            500
        ); //wBTC: 50% reserve change versus 60% reserve change
        console.log(lastReserves1[0]); //14999999999
        assertEq(
            ((lastReserves0[1] - lastReserves1[1]) * 1000) / las
            333
        ); //wETH: 33% reserve change versus 37.5% reserve chang
        console.log(lastReserves1[1]); //1066666666666666667199
        assertApproxEqAbs(lastReserves1[2], 96000e4, 1); //USDC:
        assertApproxEqAbs(lastReserves1[3], 101000e4, 1); //USDT
    }
...
```

As seen in POC, USDT/USDC reserve changes of more than 3% are allowed, whereas WBTC/WETH reserve changes are restricted. The test is based on 50% for max percentage per block increase and 33% max percentage per block decrease. Although the exact percentage change settings can be different and the actual token reserve changes per block differ by assets, the idea is such universal cap value likely will not fit multiple tokens or trading pairs. The cap can be either set too wide which is then no need to have a cap at all, or be set in a manner that restricts some volatile token assets.

See test results.

```
Running 1 test for test/pumps/Pump.HardCap.t.sol:PumpHardCapTest
[PASS] test_HardCap() (gas: 489346)
Logs:
  14999999999
  1066666666666666667199
```

```
Test result: ok. 1 passed; 0 failed; finished in 18.07ms
```

Tools Used

VS Code.

Recommended Mitigation Steps

Different assets have different levels of volatility, and such variations in volatility are typically taken into account in an AMM or oracle. (Think UniswapV3 with different tick spacings, or chainlink with different price reporting intervals).

It's better to avoid setting up `LOG_MAX_INCREASE`, `LOG_MIN_INCREASE` directly in MultiFlowPump.sol.

(1) Instead, in Well.sol allow `MAX_INCREASE` and `MAX_DECREASE` to be token specific and included as part of the immutable data created at the time of Well.sol development from Aquifer.sol, such that a token address can be included together with `MAX_INCREASE` and `MAX_DECREASE` as immutable constants.

(2) Then in `_updatePumps()`, pass current token reserves, and pass token `MAX_INCREASE` and `MAX_DECREASE` as `_pump.data` to MultiFlowPump.sol `update()`.

(3) In MultiFlowPump.sol `update()`, when calculating `_capReserve()`, use decoded `_pump.data` to pass `MAX_INCREASE`, `MAX_DECREASE` for token specific cap calculation.

publiuss (Basin) disagreed with severity and commented:

> A separate Beanstalk Pump could always be deployed. Plus, the effects of manipulation are the same whether it is volatile or stable. Recommend changing to low.

alcueca (Judge) commented:

> The documentation provided makes no mention that the all the tokens in MultiFlowPump should have similar typical volatility. The MultiFlowPump is vulnerable exactly as described.

[M-13] Useless Modifier and Bad Assumptions in Constructor

Submitted by [erebus]

Useless modifier in consructor here (even the constructor is empty, just why the modifier there, it makes no sense)

Proof of Concept

Bad assumptions to calculate the number of block passed here. BLOCK_TIME is an immutable variable which is passed as a parameter to the constructor. However, with future updates to the Ethereum protocol (IDK, reth or other validation algorithm) which could reduce that number would mean that functions dependant on that to work incorrectly like readInstantaneousReserves or _readCumulativeReserves.

Mitigation Steps

Do not assume the block rate to remain constant in the future or implement some function that modifies the BLOCK_TIME every T seconds/minutes/whatever.

[alcueca (Judge) increased severity to Medium]

[publiuss (Basin) acknowledged and commented]

> First QA item is acknowledged. The modifier is there to initialize the `ReentrancyGuard` contract. Imo, its cleaner to have an empty constructor than to have no constructor and have the user realize that it uses the `ReentrancyGuard` modifier by defualt. This item is a duplicate of other QA reports.

> Second QA item is a duplicate of: #176. See comment on issue for remediation status.

Low Risk and Non-Critical Issues

For this audit, 56 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xprinc** received the top score from the judge.

The following wardens also submitted reports: 33audits, codegpt, Trust, te_aut, alexzoid, seth_lawson, Deekshith99, radev_sw, Inspecktor, Kaysoft, peanuts, CRIMSON-RAT-REACH, QiuhaoLi, 0xWaitress, josephdara, qpzm, sces60107, kutugu, pfapostol, Topmark, hunter_w3b, bigtone, 2997ms, 0xAnah, glcanvas, LosPollosHermanos, TheSavageTeddy, Eeyore, max10afternoon, a3yip6, mahdirostami, JGcarv, kaveyjoe, oakcobalt, ptsanev, ginlee, zhaojie, 0xkazim, DanielWang888, ziyou-, erebus, 8olidity, Udsen, 0x11singh99, Eurovickk, CyberPunks, twcctop, John, Qeew, 404Notfound, MohammedRizwan, Rolezn, Jorgect, ravikiranweb3, *and* fatherOfBlocks.

1. No need to write `Aquifer.sol/wellImplementation()` getter function instead make the mapping `wellImplementations` as a public state variable.

wellImplementations : https://github.com/code-423n4/2023-07-basin/blob/main/src/Aquifer.sol#L25 Aquifer.sol/wellImplementation() : https://github.com/code-423n4/2023-07-basin/blob/main/src/Aquifer.sol#L86

2. No need to introduce a new local variable in this function `Well.sol/tokens()` , directly return the tokens array.

This is the optimised code

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L84

```
    function tokens() public pure returns (IERC20[] memory) {
        return _getArgIERC20Array(LOC_VARIABLE, numberOfTokens()
    }
```

3. `Well.sol/_swapFrom()` function using the some lines of

code same as written in another function `getSwapOut()`, which is increasing the codesize and deployment cost, instead call the function `getSwapOut()` inside the `_swapFrom()`

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L222C9-L228C80

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L246C9-L253C92

4. `Well.sol/swapTo()` function using the some lines of code same as written in another function `getSwapIn()`, which is increasing the codesize and deployment cost, instead call the function `getSwapIn()` inside `swapTo()`

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L272C9-L278C80

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L312C9-L318C91

5. Function `Well.sol/_getIJ()` will always revert for two equal token addresses due to an if-else condition.

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L742

Rather change the `else if` to `if`

6. In `MultiFlowPump.sol/update()` use `slot.readLastReserves()` to get the value of `numberOfReserves` instead of using `reserves.length`

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L73C9-L80C87

A better version of `update()` function can be this

```
function update(uint256[] calldata reserves, bytes calldata)
    PumpState memory pumpState;

    // All reserves are stored starting at the msg.sender ac
    bytes32 slot = _getSlotForAddress(msg.sender);

    // Read: Last Timestamp & Last Reserves
    (uint256 numberOfReserves , pumpState.lastTimestamp, pum
```

7. The non-zero reserve condition is used in `MultiFlowPump.sol/_init()` and `MultiFlowPump.sol/update()`. Since `_init()` function is called by only `upadate()` function, this will be waste to check this same validation two times

Recommendation : remove any of these check occurrence.

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L86

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L153

8. Can use `numberOfReserves` instead of `byteReserves.length`, this saves the calculation of `bytesReserves.length` in `MultiFlowPump.sol/_init()`

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L162

9. `MultiFlowPump.sol/_capReserve()` can be set as a `pure` function since this doesn’t view any state of chain.

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L199

10. Minor typo in commenting in `MultiFlowPump.sol/_capReserve()`

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L210

11. Better to declare the return variable as `emaReserves` instead of `reserves` as it gives more information about the return value in `MultiFlowPump.sol/readLastInstantaneousReserves()`

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L222

12. updating `lastReserves[i]` by using `_capReserves()` is not consistent in functions `MultiFlowPump.sol/readInstantaneousReserves()` and `MultiFlowPump.sol/update()`

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L119

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L256

13. Better to declare the return variable as `cumulativeReserves` instead of `reserves` as it gives more information about the return value in `MultiFlowPump.sol/readLastCumulativeReserves()`

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L267

14. No need to introduce a new local variable in this function `MultiFlowPump.sol/readCumulativeReserves()` , directly return the tokens array.

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L280

directly return using

```
        return abi.encode(byteCumulativeReserves);
```

[publiuss (Basin) confirmed and commented](#):

> For reference:

1. No remediation
2. No remediation
3. No remediation
4. No remediation
5. Added Documentation
6. No remediation
7. Addressed
8. Somewhat Addressed
9. No remediation
10. Addressed
11. Addressed
12. No remediation - This is intentional
13. Addressed
14. No remediation

Gas Optimizations

For this audit, 27 reports were submitted by wardens detailing gas optimizations. The [report highlighted below] by **SM3_SS** received the top score from the judge.

The following wardens also submitted reports: Raihan, 0xprinc, JCN, seth_lawson, bigtone, Isaudit, TheSavageTeddy, Rolezn, oakcobalt, 0xn006e7, K42, SY_S, ElCid, 0x11singh99, 0xSmartContract, peanuts, josephdara, hunter_w3b, pfapostol, SAAJ, DavidGiladi, 0xAnah, mahdirostami, MIQUINHO, Strausses, *and* erebus.

[G-01] Access mappings directly rather than using accessor functions

When you have a mapping, accessing its values through accessor functions involves an additional layer of indirection, which can incur some gas cost. This is because accessing a value from a mapping typically involves two steps: first, locating the key in the mapping, and second, retrieving the corresponding value.

```
File: /src/Aquifer.sol
87    return wellImplementations[well];
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Aquifer.sol#L87

[G-02] public functions not called by the contract should be declared external instead

when a function is declared as public, it is generated with an internal and an external interface. This means the function can be called both internally (within the contract) and externally (by other contracts or accounts). However, if a public function is never called internally and is only expected to be invoked externally, it is more gas-efficient to explicitly declare it as external.

```
File: /src/Well.sol
31    function init(string memory name, string memory symbol) pu
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L31

```
File: /src/pumps/MultiFlowPump.sol
222    function readLastInstantaneousReserves(address well) publ

239     function readInstantaneousReserves(address well, bytes m

267     function readLastCumulativeReserves(address well) public

280     function readCumulativeReserves(address well, bytes memo

307   function readTwaReserves(
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L222

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L239

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L267

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L280

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L307

[G-03] Amounts should be checked for 0 before calling a transfer

It can be beneficial to check if an amount is zero before invoking a transfer function, such as transfer or safeTransfer, to avoid unnecessary gas costs associated with executing the transfer operation. If the amount is zero, the transfer operation would have no effect, and performing the check can prevent unnecessary gas consumption.

```
File: /src/Well.sol
370   tokenOut.safeTransfer(recipient, amountOut);

447   _tokens[i].safeTransfer(recipient, tokenAmountsOut[i]);
```

```
512    tokenOut.safeTransfer(recipient, tokenAmountOut);

558    _tokens[i].safeTransfer(recipient, tokenAmountsOut[i]);

780    token.safeTransferFrom(from, address(this), amount);
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L370

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L447

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L512

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L558

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L780

[G-04] With assembly, .call (bool success) transfer can be done gas-optimized

Return data (bool success,) has to be stored due to EVM architecture, but in a usage like below, ‘out’ and ‘outsize’ values are given (0,0), this storage disappears and gas optimization is provided. (bool success,) = dest.call{value:amount}(""); bool success; assembly {
success := call(gas(), dest, amount, 0, 0) }

```
File: /src/Aquifer.sol
55   (bool success, bytes memory returnData) = well.call(initFur
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Aquifer.sol#L55

[G-05] Use constants instead of type(uintx).max

using type(uintx).max can result in higher gas costs because it involves a runtime operation to calculate the maximum value at runtime. This calculation is performed every time the expression is evaluated.

To save gas, it is recommended to use constants instead of type(uintx).max to represent the maximum value. By declaring a constant with the maximum value, the value is known at compile-time and does not require any runtime calculations.

```
File: /src/libraries/LibBytes.sol
40  require(reserves[0] <= type(uint128).max, "ByteStorage: too

41  require(reserves[1] <= type(uint128).max, "ByteStorage: too

49  require(reserves[2 * i] <= type(uint128).max, "ByteStorage:

50  require(reserves[2 * i + 1] <= type(uint128).max, "ByteStora

63  require(reserves[reserves.length - 1] <= type(uint128).max,
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibBytes.sol#L40

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibBytes.sol#L41

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibBytes.sol#L49

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibBytes.sol#L50

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibBytes.sol#L63

[G-06] Duplicated require()/if() checks should be refactored to a modifier or function

It is common to use require() or if() statements to validate certain conditions before executing specific code. However, when the same checks are repeated multiple times within a contract, it can result in redundant code and unnecessary gas consumption. To save gas and improve code readability and maintainability, it is recommended to refactor duplicated checks into modifiers or functions. By doing so, the checks can be abstracted into reusable code blocks that can be applied to multiple functions within the contract.

```
File: /src/pumps/MultiFlowPump.sol
225  if (numberOfReserves == 0) {

243  if (numberOfReserves == 0) {

270  if (numberOfReserves == 0) {

289  if (numberOfReserves == 0) {
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L225

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L243

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L270

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L289

```
File: /src/Aquifer.sol
41  if (salt != bytes32(0)) {

47  if (salt != bytes32(0)) {
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Aquifer.sol#L41

https://github.com/code-423n4/2023-07-basin/blob/main/src/Aquifer.sol#L47

```
File: /src/libraries/LibContractInfo.sol
19  if (success) {

37  if (success) {
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibContractInfo.sol#L19

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibContractInfo.sol#L37

[G-07] x += y costs more gas than x = x + y for state variables

```
File: /src/Well.sol
103    dataLoc += PACKED_ADDRESS;

105    dataLoc += ONE_WORD;
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L103

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L105

[G-08] Remove the initializer modifier

If we can just ensure that the initialize() function could only be called from within the constructor, we shouldn't need to worry about it getting called again.

```
File: /src/Well.sol
31     function init(string memory name, string memory symbol) pu
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L31

[G-09] Replace state variable reads and writes within loops with local variable reads and writes.

When accessing state variables within loops, each read or write operation incurs additional gas costs due to the storage and memory access operations involved. These costs can accumulate quickly, particularly in loops with a large number of iterations.

```
File: /src/Well.sol
101         for (uint256 i; i < _pumps.length; i++) {
            _pumps[i].target = _getArgAddress(dataLoc);
            dataLoc += PACKED_ADDRESS;
            pumpDataLength = _getArgUint256(dataLoc);
            dataLoc += ONE_WORD;
            _pumps[i].data = _getArgBytes(dataLoc, pumpDataLengt
            dataLoc += pumpDataLength;
        }
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L101-L108

[G-10] Multiple accesses of a mapping/array should use a storage pointer

Caching a mapping's value in a storage pointer when the value is accessed multiple times saves ~40 gas per access due to not having to perform the same offset calculation every time. Help the Optimizer by saving a storage variable's reference instead of repeatedly fetching it.

To achieve this, declare a storage pointer for the variable and use it instead of repeatedly fetching the reference in a map or an array. As an example, instead of repeatedly calling _pumps[i], save its reference via a storage pointer: _pumpsInfo storage pumpsInfo = _pumps[i] and use the pointer instead.

Cache storage pointers for _pumps[i]

```
file:    libraries/LibWellConstructor.sol

36          function encodeWellImmutableData(
        address _aquifer,
        IERC20[] memory _tokens,
        Call memory _wellFunction,
        Call[] memory _pumps
    ) internal pure returns (bytes memory immutableData) {
        bytes memory packedPumps;
        for (uint256 i; i < _pumps.length; ++i) {
            packedPumps = abi.encodePacked(
                packedPumps,            // previously packed pun
```

```
                _pumps[i].target,        // pump address
                _pumps[i].data.length,   // pump data length
                _pumps[i].data           // pump data (bytes)
            );
```

```
74          name = string.concat(name, ":", LibContractInfo.getSy
             symbol = string.concat(symbol, LibContractInfo.getSy
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibWellConstructor.sol#L36-L50

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibWellConstructor.sol#L74-L75

[G-11] Using bools for storage incurs overhead

```
// Booleans are more expensive than uint256 or any type that tak
// word because each write operation emits an extra SLOAD to fir
// slot's contents, replace the bits taken up by the boolean, ar
// back. This is the compiler's defense against contract upgrade
// pointer aliasing, and it cannot be disabled.
```

Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (20000 gas) when changing from false to true, after having been true in the past.

```
file:

417     bool feeOnTransfer

735     bool foundI = false;

736     bool foundJ = false;
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L417

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L735

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L736

```
file:  src/Aquifer.sol

55     (bool success, bytes memory returnData) = well.call(initF
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Aquifer.sol#L55

```
file:    src/libraries/LibContractInfo.sol

17      (bool success, bytes memory data) = _contract.staticcall

35      (bool success, bytes memory data) = _contract.staticcall

53      (bool success, bytes memory data) = _contract.staticcall
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibContractInfo.sol#L17

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibContractInfo.sol#L35

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibContractInfo.sol#L53

[G-12] String literals passed to abi.encode()/abi.encodePacked() should not be split by commas

String literals can be split into multiple parts and still be considered as a single string literal. Adding commas between each chunk makes it no longer a single string, and instead multiple strings. EACH new comma costs 21 gas due to stack operations and separate MSTOREs.

```
file:   src/libraries/LibWellConstructor.sol
```

```
81      initFunctionCall = abi.encodeWithSignature("init(string,
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibWellConstructor.sol#L81

[G-13] Using a positive conditional flow to save a NOT opcode

Estimated savings: 3 gas

```
file:   src/Aquifer.sol

56      if (!success)
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Aquifer.sol#L56

```
file:     src/Well.sol

748       if (!foundI) revert InvalidTokens();
749       if (!foundJ) revert InvalidTokens();
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L748

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L749

[G-14] Call block.timestamp direclty instead of function

```
file:   src/pumps/MultiFlowPump.sol

251     uint256 deltaTimestamp = _getDeltaTimestamp(lastTimestam

297     uint256 deltaTimestamp = _getDeltaTimestamp(lastTimestam
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L251 https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L297

[G-15] Using fixed bytes is cheaper than using string

As a rule of thumb, use bytes for arbitrary-length raw byte data and string for arbitrary-length string (UTF-8) data. If you can limit the length to a certain number of bytes, always use one of bytes1 to bytes32 because they are much cheaper.

```
file: src/functions/ConstantProduct2.sol

69    function name() external pure override returns (string mem
        return "Constant Product 2";
    }


73    function symbol() external pure override returns (string m
        return "CP2";
    }
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/functions/ConstantProduct2.sol#L69-L71

https://github.com/code-423n4/2023-07-basin/blob/main/src/functions/ConstantProduct2.sol#L73-L75

```
File: src/Aquifer.sol

62                  revert InitFailed(abi.decode(returnData, (stri
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Aquifer.sol#L62

```
file: src/libraries/LibContractInfo.sol

16    function getSymbol(address _contract) internal view returr
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibContractInfo.sol#L16

[G-16] Not using the named return variables when a function returns, wastes deployment gas

```
file: /src/pumps/MultiFlowPump.sol

350         return uint256(uint40(block.timestamp) - lastTimestan
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L350

```
file: /src/Well.sol

649             return reserves;

762             return j;
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L649

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L762

```
file: /src/libraries/LibBytes.sol

88             return reserves;
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/libraries/LibBytes.sol#L88

Recommended Code

```
    return (uint40(block.timestamp) - lastTimestamp);
```

[G-17] Use assembly for math (add, sub, mul, div)

Use assembly for math instead of Solidity. You can check for overflow/underflow in assembly to ensure safety. If using Solidity versions < 0.8.0 and you are using Safemath, you can gain significant gas savings by using assembly to calculate values and checking for overflow/underflow.

```
file: /src/pumps/MultiFlowPump.sol

343         return ((numberOfReserves - 1) / 2 + 1) << 5;
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/pumps/MultiFlowPump.sol#L343

```
file: /src/functions/ConstantProduct2.sol

65         reserve = lpTokenSupply ** 2;

99         reserve = reserves[i] * ratios[j] / ratios[i];
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/functions/ConstantProduct2.sol#L65

https://github.com/code-423n4/2023-07-basin/blob/main/src/functions/ConstantProduct2.sol#L99

```
file: /src/functions/ProportionalLPToken2.sol

22         underlyingAmounts[0] = lpTokenAmount * reserves[0] / l
23         underlyingAmounts[1] = lpTokenAmount * reserves[1] / l
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/functions/ProportionalLPToken2.sol#L22

https://github.com/code-423n4/2023-07-basin/blob/main/src/functions/ProportionalLPToken2.sol#L23

```
file: /src/Well.sol

90        uint256 dataLoc = LOC_VARIABLE + numberOfTokens() * O

98        uint256 dataLoc = LOC_VARIABLE + numberOfTokens() * O

176        uint256 pumpDataLength = _getArgUint256(dataLoc + PA

232        amountOut = reserveJBefore - reserves[j];

282        amountIn = reserves[i] - reserveIBefore;
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L90

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L98

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L176

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L232

https://github.com/code-423n4/2023-07-basin/blob/main/src/Well.sol#L282

[G-18] Refactor event to avoid emitting empty data

```
file: /src/Aquifer.sol

58                if (returnData.length < 68) revert InitFailed
```

https://github.com/code-423n4/2023-07-basin/blob/main/src/Aquifer.sol#L58

publiuss (Basin) acknowledged and commented:

> G-01 No remediation
> G-02 Fixed
> G-03 No remediation - costs more gas
> G-04 No remediation
> G-05 Fixed
> G-06 No remediation - costs more gas

> G-07 No remediation - costs more gas
> G-08 No remediation - the initializer modifier is necessary.
> G-09 No remediation - costs more gas
> G-10 No remediation - This is in a script and not in a contract.
> G-11 Fixed
> G-12 No remediation - This is in a script and not in a contract.
> G-13 No remediation - This code has already been removed.
> G-14 No remediation
> G-15 No remediation
> G-16 No remediation - costs more gas
> G-17 No remediation
> G-18 No remediation

Audit Analysis

For this audit, 7 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **Trust** received the top score from the judge.

The following wardens also submitted reports: peanuts, 0xSmartContract, Eeyore, oakcobalt, glcanvas, and K42.

General

I've spent 2-3 days on this audit and covered the code in scope. Having been audited twice by private firms, I expected there to be very little in terms of low-hanging fruits.

Approach

Most of the focus was on deeply understanding the assembly optimizations and particular LP mechanics, informally proving their correctness and so on. Differential analysis between Basin and Uniswap was helpful for this task.

Architecture recommendations

The platform supports a very generic well implementation, and as the team understands, this leads to a broad variety of malicious deployed well risks. It should be a high priority task for the team to find good ways of representing all Well trust assumptions to its users, and expose that through a smart contract UI or an open source front-end library.

Qualitative analysis

Code quality and documentation is very mature. The test suite is pretty comprehensive and fuzz tests are a great way to complement the static tests. My suggestion is to add integration tests to verify behavior of the system as a whole, rather than all its specific sub-components.

Centralization risks

None. The architecture is fully permissionless.

Systemic risks

MEV and TWAP manipulation are the main systemic risks. Interacting with Wells registered in the Aquifer could possibly be risky, depending on the well's configuration.

The immutability of the Pump and Well co-efficients, such as LOG*MAX*INCREASE and ALPHA, present a systemic risk as time progresses and the optimal values start diverging.

As the entire platform is unupgradeable, migration in the event of a security hole or a black swan event will be challenging. The team should prepare multiple recovery scenarios and set up adequate action channels to prepare for such eventualities.

Time spent:

25 hours

publiuss (Basin) acknowledged

alcueca (Judge) commented:

> While other reports have included much more text, this report in particular stands out for the quality of the advice given.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.