2024-04-Beanstalk-DIB/audit-data/2024-04-Beanstalkreport.md

Title: 04-2024-Beanstalk Audit Report author: Idriis Perrin date: April 28,2024

Prepared by: [Idriis Perrin] Lead Auditor:

- Idriis Perrin

Protocol Summary

Basin is a composable EVM-native decentralized exchange protocol that allows for the composition of arbitrary exchange functions, network-native oracles and exchange implementations into a single liquidity pool known as a Well. In practice, Basin lowers the friction for market makers to deploy liquidity with custom orders and allows their liquidity to be used by other network-native protocols without additional trust assumptions.

A Well is a constant function AMM that allows the provisioning of liquidity into a single pooled on-chain liquidity position. Each Well is defined by its Tokens, Well Function, and Pump:

The Tokens define the set of ERC-20 tokens that can be exchanged in the Well. The Well Function defines an invariant relationship between the Well's reserves and the supply of LP tokens. Pumps are on-chain oracles that are updated upon each interaction with the Well. Multi Flow Pump is an inter-block MEV manipulation resistant network-native Pump implementation for arbitrary current data in an EVM for both instantaneous and time-weighted average values.

Disclaimer

I guarantee to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for 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 and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		IMPACT		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

```
src/
└── contracts/
    ├── functions/
    │   ├── ConstantProduct2.sol
    │   └── ProportionalLPToken2.sol
    ├── libraries/
    │   ├── LibLastReserveBytes.sol
    │   └── LibMath.sol
    └── pumps/
        └── MultiFlowPump.sol
```

Roles

The only "actors" in the context of Basin are traders (takers) and liquidity providers (makers), as with any decentralized exchange. In a sense, external protocols that consume data stored in Pumps (like Multi Flow) could be considered another actor.

Issues found

SEVERITY	NUMBER OF ISSUES FOUND
High	1
Medium	0
Low	3
Info	0
Total	4

Findings

High

[H-1] Unprotected initializer (access control)

Description:

- The function `MultiFlowPump::_init` lacks proper access control or safety measures to ensure it is called only once or by authorized parties.

- The function `MultiFlowPump::isInitialized` is not properly secured against being executed multiple times . `MultiFlowPump::_init` can be called by unauthorized parties.

Impact: Unprotected initializers can lead to security vulnerabilities such as reinitialization attacks or unexpected contract states. If the initialization logic sets critical state variables or performs sensitive operations, multiple calls to the initializer could compromise the integrity of the contract. Multiple invocations of the initializer function can result in unexpected behavior within the contract. This could lead to inconsistencies in the contract's state or functionality, potentially causing financial losses or other adverse effects for users interacting with the contract.

Unprotected initializers can introduce operational risks for developers and users of the contract. If the contract relies on specific initialization conditions or assumes that initialization only occurs once, multiple invocations of the initializer could disrupt normal contract operations and lead to operational inefficiencies or failures.

Proof of concept:

The _init function can be called multiple times, potentially leading to unexpected behavior or security vulnerabilities if the contract's logic relies on _init being called only once.

```

pragma solidity 0.8.0;

contract UnprotectedInitializer {
    bytes32 public slot;
    uint40 public lastTimestamp;
    uint256[] public reserves;

    // Unprotected initializer function
    function _init(bytes32 _slot, uint40 _lastTimestamp, uint256[] memory _reserves) internal {
        slot = _slot;
        lastTimestamp = _lastTimestamp;
        reserves = _reserves;
    }
}

contract Attacker {
    UnprotectedInitializer target;

    constructor(UnprotectedInitializer _target) {
        target = _target;
    }

    function attack() external {
        // Calling the _init function of the target contract
        uint256[] memory reserves = new uint256[](3);
        reserves[0] = 100;
        reserves[1] = 200;
        reserves[2] = 300;

        target._init(0x123456789abcdef, 1234567890, reserves);

        //  calling _init again with different values
        uint256[] memory newReserves = new uint256[](2);
        newReserves[0] = 400;
        newReserves[1] = 500;

        target._init(0xabcdef123456789, 9876543210, newReserves);
    }
}
```

Recomended mitigation:

Restrict access to the _init function to only the intended callers. You can use access modifiers like internal or private to limit access to the function within the contract or a designated set of contracts.

```
modifier onlyOwner {
    require(msg.sender == owner, "Only owner can call this function");
    _;
}

function _init(bytes32 slot, uint40 lastTimestamp, uint256[] memory reserves) internal onlyOwner {
    // Initialization logic
}
```

Low

[L-1] public functions not used internally could be marked external

Found in src/libraries/LibLastReserveBytes.sol Line: 59

Description: The function can only be called from outside the contract. It cannot be called internally or via another contract. Also external functions have a cheaper gas cost for calling as compared to public functions. This is because calling an external function involves a jump to a different context (i.e., the called contract), whereas calling a public function can be done within the same context.

Impact:

Marking public functions that are not used internally as "external" in Solidity can have several impacts, both in terms of code clarity and gas optimization

Proof of concept:

```
pragma solidity ^0.8.0;

contract ExContract {
    uint256 public maxI;
    uint256[] public reserves;

    constructor(uint256 _maxI) {
        maxI = _maxI;
        reserves.push(0); // Initialize reserves array with a placeholder value
    }

    // External function to interact with reserves array
    function getReserveAtIndex(uint256 index) external view returns (uint256) {
        require(index < reserves.length, "Index out of bounds");
        return reserves[index];
    }

    // Example internal function used within the contract
    function doSomethingInternal(uint256 value) internal {
        // Function implementation
    }

    // Function with assembly block (internal to the contract)
    function manipulateStorage() internal {
        uint256 iByte;
        for (uint256 i = 1; i < maxI; ++i) {
            iByte = i * 64;
            assembly {
                sstore(
                    add(slot, i),
                    add(mload(add(reserves, add(iByte, 32))), shr(128, mload(add(reserves, add(iByte, 64)))
                )
            }
        }
    }
}
```

Recomended mitigation:

For public functions that are not used internally but are part of the contract's external interface, update their visibility modifiers from "public" to "external". This change clearly defines the contract's boundaries and optimizes gas usage.

[L-2] Literal variables being used instead of constant.

LibMath.sol

Description:

Using constant variables in Solidity provides clarity and maintainability to your code. It allows you to define and reuse values that remain constant throughout the contract, improving readability and making it easier to update these values in the future.

Proof of concept:

1. Found in src/libraries/LibMath.sol Line: 44
2. Found in src/libraries/LibMath.sol Line: 46
3. Found in src/libraries/LibMath.sol Line: 58

```


        uint256 a0 = (10 ** n) * a;


        uint256 xNew = 10;
        root = (xNew + 5) / 10;
```

Recomended mitigation:

Use Constants: Define constants for values that do not change throughout the contract's lifecycle. Constants can be declared using constant keyword or through immutable variables with constant values. Centralize Constants: If multiple contracts within your system use the same constant values, consider centralizing those constants in a single contract or a library. This promotes consistency and reduces redundancy.

[L-3] Floating pragma

Description:

In Solidity, the pragma directive is used to specify the version of the Solidity compiler to be used for compiling the contract. The pragma statement is typically placed at the beginning of a Solidity source file.

A "floating pragma" refers to a specific form of the pragma directive where only the major version of the compiler is specified, while the minor and patch versions are left unspecified.

Impact: Using a floating pragma exposes contracts to potential security risks associated with new compiler versions. While newer versions may include security improvements or bug fixes, they may also introduce new vulnerabilities or behavioral changes that could affect contract security. Developers must carefully review compiler release notes and assess the impact of upgrading to newer versions on contract security.

Proof of concept:

- Found in src/libraries/LibLastReserveBytes.sol Line: 3
- Found in src/functions/ProportionalLPToken2.sol Line: 3

```
pragma solidity ^0.8.20;
```

Recomended mitigation: The pragma of the contract should be solid.