symmio-v0-8-4 Protocol Audit Report

Prepared by: Hamid

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
- Executive Summary
  - Issues found
- Findings
- High
  - [H-1] RE-ENTRANCY attack in `Accout:AccountFacetImpl.sol` allow malicious actor to drain funds
  - [H-2] Local var `numFacet` in the `DiamondLoupFacet.sol:facets` function is set to zero, desrupting the total logic of the code.
- Medium
  - [M-1] DOS (denial of service) vulnerability in the nested loop within the `DiamondLoupFacet.sol:facets` function.
- Low
- Informational
  - [I-1] MAGIC NUMBER. Less cleaner and prone to error
  - [G-1] Use an if/revert statement instead of require
- Gas
  - [G-2] --I over I--

Protocol Summary

Protocol does X, Y, Z

Disclaimer

The YOUR_NAME_HERE team makes all effort 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

		Impact		
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

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

Audit Details

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	0
Info	8
Total	0

Findings

High

[H-1] RE-ENTRANCY attack in `Accout:AccountFacetImpl.sol` allow malicious actor to drain funds

Description: The `Accout:AccountFacetImpl.sol:deposit` function does not follow CEI(check, effect and interaction) as a result of that any bad actor/contract can drain the funds.

```
    // @audit re entrancy attack, function does not follow CEI
    // impact high
    // likelihood is high
    function deposit(address user, uint256 amount) internal {
        GlobalAppStorage.Layout storage appLayout =
GlobalAppStorage.layout();
>@      IERC20(appLayout.collateral).safeTransferFrom(msg.sender,
address(this), amount);
        // use a constant instead
        uint256 amountWith18Decimals = (amount * 1e18) / (10 **
IERC20Metadata(appLayout.collateral).decimals());
```

```
>@      AccountStorage.layout().balances[user] += amountWith18Decimals;
    }
```

A user can call the `Accout:AccountFacetImpl.sol:deposit` function with a milicious contract that contain the `receive/fallback` function and potentially drain all funds **Impact:** possible to for a bad actor to drain all funds from the `Accout:AccountFacetImpl.sol` contract **Proof of Concept:**

1. multiple users call the `Accout:AccountFacetImpl.sol:deposit`functions
2. bad actor set up a `fallback/receive` contract.
3. bad actor call the `Accout:AccountFacetImpl.sol:deposit` function

```
    contract ReentrancyAttack {
    AccoutFacetImpl public accoutFacet;
    address public attacker;

    constructor(AccoutFacetImpl _accoutFacet) {
        accoutFacet = _accoutFacet;
        attacker = msg.sender;
    }

    receive() external payable {
        if (address(accoutFacet).balance >= 1 ether) {
            accoutFacet.deposit(attacker, 1 ether);
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether, "Insufficient ETH");
        accoutFacet.deposit{value: 1 ether}(address(this), msg.value);
    }
}

contract ReentrancyTest is DSTest {
    AccoutFacetImpl accoutFacet;
    ReentrancyAttack reentrancyAttack;
    IERC20 collateralToken;

    function setUp() public {
        accoutFacet = new AccoutFacetImpl();
        collateralToken = IERC20(address(accoutFacet.collateral())); //
Assuming collateral is an ERC20 token
        collateralToken.mint(address(this), 100 ether); // Mint tokens for
testing
        reentrancyAttack = new ReentrancyAttack(accoutFacet);
    }

    function testReentrancyAttack() public {
        collateralToken.approve(address(accoutFacet), 10 ether); //
Approve tokens for deposit
        accoutFacet.deposit(address(this), 10 ether); // Initial deposit
```

```
        uint256 initialBalance = collateralToken.balanceOf(address(this));
        reentrancyAttack.attack{value: 1 ether}(); // Launch the attack

        uint256 finalBalance = collateralToken.balanceOf(address(this));
        assertTrue(finalBalance > initialBalance, "Reentrancy attack
failed to drain funds");
    }
}
```

Recommended Mitigation:

1. dont be shy to smash non reenctrancy from OZ library to `Accout:AccountFacetImpl.sol:deposit` function
2. follow CEI methodology/Pattern

```
```solidity
```

- ```
    import {ReentrancyGuard} from
  "@openzeppelin/contracts/security/ReentrancyGuard.sol";
   //----code conination-----//
    function deposit(address user, uint256 amount) internal
  nonReentrant{
  ```
- ```
        // checks
  ```
- ```
        if (user == address(o)) {
  ```
- ```
            revert zeroAddr();
  ```
- ```
       }
  ```
- ```
        if (amount == 0) {
  ```
- ```
            revert zeroValue();
  ```

- ```
        }
    GlobalAppStorage.Layout storage appLayout =
  GlobalAppStorage.layout();
  ```
- ```
    // @audit Magic numbers... 1
    // use a constant instead
    uint256 amountWith18Decimals = (amount * 1e18) / (10 **
  IERC20Metadata(appLayout.collateral).decimals());
  ```
- ```
    IERC20(appLayout.collateral).safeTransferFrom(msg.sender,
  address(this), amount);
  ```
- ```
     AccountStorage.layout().balances[user] += amountWith18Decimals;
  ```
- ```
    IERC20(appLayout.collateral).safeTransferFrom(msg.sender,
  address(this), amount);
  ```
- ```
   AccountStorage.layout().balances[user] += amountWith18Decimals;
  ```

  }

  ```

  ```

[H-2] Local var `numFacet` in the `DiamondLoupFacet.sol:facets` function is set to zero, desrupting the total logic of the code.

Description: Identified a vulnerability in the facets function where the variable numFacets is initialized to zero but is subsequently used in loop logic. This leads to the function logic being disrupted and not functioning as intended.

Impact:

Proof of Concept: The vulnerability arises because numFacets is set to zero initially and is used in a loop condition. Without proper initialization or assignment, the loops relying on numFacets fail to execute correctly, breaking the function's intended logic.

```

```

```
    uint256 numFacets; // Initialized to zero
// Further code relies on numFacets
for (uint256 facetIndex = 0; facetIndex < numFacets; facetIndex++) {
    // Loop logic
}
```

Recommended Mitigation:

```
    function facets() external view override returns (Facet[] memory
facets_) {
        LibDiamond.DiamondStorage storage ds =
LibDiamond.diamondStorage();
        uint256 selectorCount = ds.selectors.length;
        // create an array set to the maximum size possible
                    facets_ = new Facet[](selectorCount);
        // create an array for counting the number of selectors for each
facet
        uint8[] memory numFacetSelectors = new uint8[](selectorCount);
        // total number of facets
        // @audit numFacets not initialised, breaks the current function
        // @mitigation creaate a numFacet to be equal to facets_.length
-       uint256 numFacets;
+       uint256 numFacets = facets_.length; // Properly initialize
numFacets
```

Medium

[M-1] DOS (denial of service) vulnerability in the nested loop within the `DiamondLoupFacet.sol:facets` function.

Description: A Denial of Service (DoS) vulnerability was identified within the `DiamondLoupFacet.sol:facets` function. This is rooted in the inefficient handling of nested loops, leading to excessive gas consumption and potential transaction failures when processing function selectors.

The vulnerability is due to the nested loop structure in the facets function. The outer loop iterates over `selectors`, and the inner loop matches each selector to the corresponding facet address.

```
for (uint256 facetIndex; facetIndex < numFacets; facetIndex++) {
                if (facets_[facetIndex].facetAddress == facetAddress_) {

facets_[facetIndex].functionSelectors[numFacetSelectors[facetIndex]] =
selector;
                    // probably will never have more than 256 functions
from one facet contract
                    require(numFacetSelectors[facetIndex] < 255);
```

```
                    numFacetSelectors[facetIndex]++;
                    continueLoop = true;
                    break;
                }
            }
            // if functionSelectors array exists for selector then
continue loop
            if (continueLoop) {
                continueLoop = false;
                continue;
            }
            // create a new functionSelectors array for selector
            facets_[numFacets].facetAddress = facetAddress_;
            facets_[numFacets].functionSelectors = new bytes4[]
(selectorCount);
            facets_[numFacets].functionSelectors[0] = selector;
            numFacetSelectors[numFacets] = 1;
            numFacets++;

        for (uint256 facetIndex; facetIndex < numFacets; facetIndex++) {
            uint256 numSelectors = numFacetSelectors[facetIndex];
            bytes4[] memory selectors =
facets_[facetIndex].functionSelectors;
            // setting the number of selectors
            assembly {
                mstore(selectors, numSelectors)
            }
        }
```

Impact: A Bad actor can exploit this vulnerability by submitting a large number of selectors, causing the function to consume excessive gas. This can lead to failed transactions, potentially disrupting the availability of the smart contract and affecting all users of the system.

Proof of Concept:

```
    function testDoSInFacets() public {
        // Initialize a large number of selectors to simulate a potential
DoS attack
        uint256 selectorCount = 1000;
        bytes4[] memory selectors = new bytes4[](selectorCount);
        address[] memory facetAddresses = new address[](selectorCount);

        for (uint256 i = 0; i < selectorCount; i++) {
            selectors[i] = bytes4(keccak256(abi.encodePacked(i)));
            facetAddresses[i] = address(this);
        }

        // Deploy facets and set selectors
        // (assuming you have a function to do this in your contract)
        yourContract.setFacets(facetAddresses, selectors);
```

```
        // Gas usage before executing the function
        uint256 gasBefore = gasleft();

        // Call the vulnerable function
        yourContract.facets();

        // Gas usage after executing the function
        uint256 gasAfter = gasleft();

        // Calculate gas used
        uint256 gasUsed = gasBefore - gasAfter;

        // Log gas usage
        emit log_named_uint("Gas used for facets function", gasUsed);

        // Assert to ensure function does not fail due to excessive gas
usage
        assert(gasUsed < block.gaslimit);
    }
```

Recommended Mitigation:

1. use mapping selectors directing to facet indexed.
2. gas limit check.
3. optimize the overall architecture (optional).
4. batch processing

```
+   //Mapping for direct access
+   mapping(address => uint256) facetIndexMap;
///code faucet////
function facets() external view override returns (Facet[] memory facets_)
{
    LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
    uint256 selectorCount = ds.selectors.length;
    facets_ = new Facet[](selectorCount);
    uint8[] memory numFacetSelectors = new uint8[](selectorCount);
    uint256 numFacets;


    for (uint256 selectorIndex = 0; selectorIndex < selectorCount;
selectorIndex++) {
        bytes4 selector = ds.selectors[selectorIndex];
        address facetAddress_ =
ds.facetAddressAndSelectorPosition[selector].facetAddress;
+       // Gas limit check
+       if (gasleft() <> MIN_GAS_LIMIT){
    revert gasLimitError();
    };

        // + Direct mapping lookup to avoid nested loop
```

```
        uint256 facetIndex = facetIndexMap[facetAddress_];
        if (facetIndex == 0 && facetAddress_ != address(0)) {
            facetIndex = ++numFacets;
            facetIndexMap[facetAddress_] = facetIndex;
            facets_[facetIndex - 1].facetAddress = facetAddress_;
            facets_[facetIndex - 1].functionSelectors = new bytes4[]
(selectorCount);
        }

        facets_[facetIndex -
1].functionSelectors[numFacetSelectors[facetIndex - 1]++] = selector;
        require(numFacetSelectors[facetIndex - 1] < 255, "Too many
functions from one facet");
-       bool continueLoop = false; // - Removed as it's no longer needed
-       // - Removed the nested loop and replaced with direct mapping
lookup
-       for (uint256 facetIndex = 0; facetIndex < numFacets; facetIndex++)
{
-           if (facets_[facetIndex].facetAddress == facetAddress_) {
-
facets_[facetIndex].functionSelectors[numFacetSelectors[facetIndex]] =
selector;
-               require(numFacetSelectors[facetIndex] < 255);
-               numFacetSelectors[facetIndex]++;
-               continueLoop = true;
-               break;
-           }
-       }
-       if (continueLoop) {
-           continueLoop = false;
-           continue;
-       }

-       facets_[numFacets].facetAddress = facetAddress_;
-       facets_[numFacets].functionSelectors = new bytes4[]
(selectorCount);
-       facets_[numFacets].functionSelectors[0] = selector;
-       numFacetSelectors[numFacets] = 1;
-       numFacets++;
    }

    for (uint256 facetIndex = 0; facetIndex < numFacets; facetIndex++) {
        uint256 numSelectors = numFacetSelectors[facetIndex];
        bytes4[] memory selectors = facets_[facetIndex].functionSelectors;
        assembly {
            mstore(selectors, numSelectors)
        }
    }

    assembly {
        mstore(facets_, numFacets)
    }
}
```

Low

Informational

[I-1] MAGIC NUMBER. Less cleaner and prone to error

Description: instead of hard writing plain numbers. use constants and immutables for a more cleaner approach

Proof of Concept: multiple scenarios of magic number being used i couldn't point then all out before i ran out of time while writing my report, but i did my best. a good example is line `BridgeFaucetImpl.sol:transferToBridge` function,`Accout:AccountFacetImpl.sol:deposit` function.

Recommended Mitigation: EXAMPLE:

```
+   uint256 private constant PRECISION = 1e18;
    //---code----//
    uint256 amountWith18Decimals = (amount * PRECISION ) / (10 **
IERC20Metadata(appLayout.collateral).decimals());
```

[G-1] Use an if/revert statement instead of require

Gas

Description: The require statement have been proven to be more gas expensive compared to an if/reevert statement. these practise happened alot in all scenerios.
`BridgeFacetImpl.sol:restoreBridgeTransaction` function line 50, etc

Proof of Concept: *EXAMPLES:*

```
    require(transactionIds[i - 1] <= bridgeLayout.lastId, "BridgeFacet:
Invalid transactionId");
```

Recommended Mitigation:

```
if (transactionIds[i - 1] > bridgeLayout.lastId) {
    revert transactionCustomError();
}
```

[G-2] --I over I--

Description: `--i` have been proven to be more efficient than `i--`

```
    for (uint256 i = transactionIds.length; i != 0; i--)
```

Impact: GAS EXPENSIVE

Recommended Mitigation:

```
for (uint256 i = transactionIds.length; i != 0; --i) //---coode----//
```