Security Review For Majora

Collaborative Audit Prepared For:	Majora
Lead Security Expert(s):	0x52
	pkqs90
	ZeroTrust
Date Audited:	November 6th - December 4th

Introduction

Majora is a next-generation decentralized finance (DeFi) yield infrastructure designed to simplify and enhance yield generation across the DeFi ecosystem. At its core, Majora enables users to create and manage ERC4626 vaults effortlessly, without requiring any coding knowledge, opening the doors to sophisticated DeFi strategies for all.

Scope

- Repository: https://github.com/majora-finance/borrow-module

  Audited commit hash: 5269ab2667327ea06c3c08e068d323bdba6e88a0

  Final commit hash: bf8fcd5d0712d0da25e8ebe5a303d7286f449d4c
- Repository: https://github.com/majora-finance/portal

  Audited commit hash: d822398fe7f07c502cad00a973c80603951c3629

  Final commit hash: 80c491150679bc17b13a10107a856e5380e66e60
- Repository: https://github.com/majora-finance/fees

  Audited commit hash: 29519f1d50c45ba357f93f656214def4631762eb

  Final commit hash: e266b938f4bac75f6e3b270c9595e354191f59ed
- Repository: https://github.com/majora-finance/mopt

  Audited commit hash: 02aa6ed856ee776c833d56aaee04ecb943729522

  Final commit hash: b4286d33f5371f24c997e1e3fa4ee4f73dd96841
- Repository: https://github.com/majora-finance/majora-blocks

  Audited commit hash: c7b420fd4e8c14ba1b4e9011d1136222ecae3f01

  Final commit hash: 62633dea34920b977688c3f30e6e4049cc5e414a
- Repository: https://github.com/majora-finance/aave-v3-blocks

  Audited commit hash: 3c5f9d91ac93a6565781be7a31859a23b6a4db31

  Final commit hash: 8a32ee94c273ae4ab1da27033816a7f679a6954b
- Repository: https://github.com/majora-finance/curve-blocks

  Audited commit hash: 46c10c95da07d7e9ad682f6e9cdc3580ca105637

  Final commit hash: a3f8b8aea6996b6f89731cf272c084c3b95e629a
- Repository: https://github.com/majora-finance/convex-blocks

  Audited commit hash: ae3b391fc7b82937680002983ecd2f5eee3a1586

  Final commit hash: 8f469f7812e3a833cc5f04cbf891536e72c26ac3

- Repository: https://github.com/majora-finance/balancer-v2-blocks

  Audited commit hash: de26336ea4dd3eff2464e9046b423468c6f0b14c

  Final commit hash: 41ca469c35b7f4c09230623a9b2a0338ded959b2
- Repository: https://github.com/majora-finance/aura-blocks

  Audited commit hash: b148cf1d9efa64bd4434d1e30ed67a34584a4a6a

  Final commit hash: ab4a1295c392dc59c6af62fc6bb1292f7004be6f
- Repository: https://github.com/majora-finance/gamma-blocks

  Audited commit hash: 962e4278fe18529a7ea6ad968c187e283b1c3569

  Final commit hash: 24c598b22f0bc64f90f6c4d2e35618f03ad7d2d5
- Repository: https://github.com/majora-finance/stargate-blocks

  Audited commit hash: 40bfd619564d14d8300ebd020bbc0146afaa3e56

  Final commit hash: 171b488edca2cea80fa1c05dd443d36ced5d94e1

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
1	28	18

Issues Not Fixed or Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: MajoraAaveV3BorrowPositionManager withdraws more collateral than expected during repay, leading to a lower health factor.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/167

Summary

When MajoraAaveV3BorrowBlock performs a repay action, the healthFactor should remain the same. However, this issue will cause the position manager to withdraw more collateral, leading to a lower health factor.

Vulnerability Detail

During repay, we want to repay a percentage amount of debtTokens, and withdraw the same percentage of collateral. This way, the health factor remains unchanged.

Let's first see the process of repaying when we don't have enough debtToken:

1. Repay a portion of the debt
2. Withdraw collateral
3. Swap collateral to debtToken
4. Finish repay
5. Withdraw collateral

The issue here is for step 5, we recalculate the amount of collateral to withdraw (total collateral times percentage) after we perform step 2. This means we would end up withdrawing more collateral than expected.

- https://github.com/sherlock-audit/2024-11-majora/blob/main/borrow-module/contracts/aave/MajoraAaveV3BorrowPositionManager.sol#L238

```
@>      uint256 amountToWithdraw =
 ↪  _position.collateral.aToken.balanceOf(address(this));
        _repay(address(_position.debt.token), amountToRepay,
 ↪  _position.debt.debtType, address(this));

        /**
         * If there is more debt token balance than repaid amount
         * swap remaining tokens to collateral token
         */
        if (!dynParamsUsed && _dynamicParams.length > 0 && debtTokenBalance >
 ↪  amountToRepay && _percent == 10000) {
            ...
        } else {
            /**
             * withdraw collateral on the vault address
             */
@>          _withdraw(address(_position.collateral.token), (amountToWithdraw *
 ↪  _percent) / 10000, msg.sender);
        }
```

Impact

Position manager will withdraw more collateral, leading to a lower than expected health factor.

Recommendation

Precalculate a ”target collateral” amount in the beginning and withdraw to the target.

Discussion

IAm0x52

Fix commit: c2c3246

Fix looks good but I have a small nitpick. It's technically incorrect in L251 to subtract `remainingCollateralAfterSwap` from the withdrawal amount, as this will leave a small amount above your target collateral.

Issue M-1: MajoraGammaQuickDepositBlock.sol should not add token's in-vault balance when calculating dynamicParamsInfo().

Source: https://github.com/sherlock-audit/2024-11-majora/issues/182

Summary

MajoraGammaQuickDepositBlock.sol should not add token's in-vault balance when calculating dynamicParamsInfo().

Vulnerability Detail

When calculating tokens used for swap, we should only use the token amount that is withdrawn from Gamma pool. If we also include the tokens inside the vault, we may be using tokens that doesn't belong to this block.

- https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/quick/MajoraGammaQuickDepositBlock.sol#L215
- https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/quick/MajoraGammaQuickDepositBlock.sol#L223

```
    if (parameters.token == parameters.token0) {
        uint256 balToken1 = IERC20(parameters.token1).balanceOf(address(this));
        swap = DataTypes.DynamicSwapParams({
            fromToken: parameters.token1,
            toToken: parameters.token,
@>          value: token1Amount + balToken1,
            valueType: DataTypes.SwapValueType.INPUT_STRICT_VALUE
        });
    } else {
        uint256 balToken0 = IERC20(parameters.token0).balanceOf(address(this));
        swap = DataTypes.DynamicSwapParams({
            fromToken: parameters.token0,
            toToken: parameters.token,
@>          value: token0Amount + balToken0,
            valueType: DataTypes.SwapValueType.INPUT_STRICT_VALUE
        });
    }
```

Impact

We may be using tokens that does not belong to this block.

Recommendation

Do not add `balToken1` and `balToken0`.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/gamma-blocks/commit/b1d02edeb7fc298462c363967efcbdf52e0464b5

Issue M-2: MajoraAaveV3BorrowPositionManagerDataAggregator may double count token balance when calculating rebalance info.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/181

Summary

MajoraAaveV3BorrowPositionManagerDataAggregator may double count token balance when calculating rebalance info.

Vulnerability Detail

When calculating the available amount of debt tokens to use for repayment, the vault balance may be double counted, because the oracleState would already include the amount of debt token balance for the vault. It should be adding the balance of the position manager instead.

https://github.com/sherlock-audit/2024-11-majora/blob/main/borrow-module/contracts/aave/MajoraAaveV3BorrowPositionManagerDataAggregator.sol#L96

```
uint256 availableAmount =
 ↪  oracleState.findTokenAmount(address(_position.debt.token)) +
 ↪  _position.debt.token.balanceOf(info.vault);
```

Impact

The rebalance execution info would be inaccurate, leading to failure of position manager rebalance.

Recommendation

Change to:

```
uint256 availableAmount =
 ↪  oracleState.findTokenAmount(address(_position.debt.token)) +
 ↪  _position.debt.token.balanceOf(_pm);
```

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/borrow-module/commit/31df64eda2f2e145b2abcf5516f5a9effed17ee1

Issue M-3: MajoraOperationsPayment Token#burn can be used to grief forwarder and waste their gas

Source: https://github.com/sherlock-audit/2024-11-majora/issues/176

Summary

`MajoraOperationsPaymentToken#burn` allows specifying any arbitrary `to` address. This can be set to a contract to waste all gas sent by the forwarder.

Vulnerability Detail

MajoraOperationsPaymentToken.sol#L162-L167

```
function burn(address _to, uint256 _amount) public {
    if (_amount == 0) revert NoBurnValue();
    _burn(_msgSender(), _amount);
    (bool sentTo,) = _to.call{value: _amount}("");
    require(sentTo, "MOPT: Error on burn");
}
```

We see above the `burn` allows specifying any arbitrary `to` address. This can be set to a contract to waste all gas sent by the forwarder.

Impact

Grief forwarder and waste their gas

Code Snippet

MajoraOperationsPaymentToken.sol#L162-L167

Tool used

Manual Review

Recommendation

Limit gas on the `call` to 50k

Discussion

IAm0x52

Fix confirmed: 1d5eea9

Issue M-4: MajoraGammaQuickDepositBlock#enter fails to check price allowing vault to enter when pool price is manipulated

Source: https://github.com/sherlock-audit/2024-11-majora/issues/173

Summary

`checkPrices` disallows the vault from entering into the pool when the prices is too far away from the reference price. This protects the vault from excess slippage and losses during deposit.

Vulnerability Detail

MajoraGammaQuickDepositBlock.sol#L233-L252

```
function enter(uint256 _index) external {
    BlockParameters memory parameters =
    ↪   abi.decode(LibBlock.getStrategyStorageByIndex(_index), (BlockParameters));
    bytes memory dynParameters = LibBlock.getDynamicBlockData(_index);
    DataTypes.DynamicSwapData memory params = abi.decode(dynParameters,
    ↪   (DataTypes.DynamicSwapData));

    uint256 amountToDeposit = (IERC20(parameters.token).balanceOf(address(this)) *
    ↪   parameters.tokenInPercent) /
        10000;

    uint256 balanceTargetTokenBefore =
    ↪   IERC20(params.targetAsset).balanceOf(address(this));

    IERC20(parameters.token).safeIncreaseAllowance(address(portal), params.amount);

    portal.majoraBlockSwap(
        params.route,
        params.approvalAddress,
        params.sourceAsset,
        params.targetAsset,
        params.amount,
        params.data
    );
```

If the spot price of the gamma pool has been manipulated or is temporarily far from market price, entering will cause the vault to receive less LP than expected for the deposited assets. This can happen either accidentally or by an attacker to drain funds from the vault.

Impact

The vault will lose an excess portion of it's funds due to slippage

Code Snippet

MajoraGammaQuickDepositBlock.sol#L233-L305

MajoraGammaUniDepositBlock.sol#L236-L307

Tool used

Manual Review

Recommendation

Call `checkPrices` during the `enter` flow.

Discussion

IAm0x52

Fix confirmed.

Quickswap commit: 8e4fbe Uniswap commit: e8bd944

Issue M-5: MajoraBalancerDepositBlock#_-checkPrice() assumes token order leading to blocks being nonfunctional for some pools

Source: https://github.com/sherlock-audit/2024-11-majora/issues/172

Summary

When calculating the price, the underlying token is always assumed to be the first non-LP token. However tokens are sorted alphabetically so this is not always the case. This will cause the rate to be checked incorrect and _checkPrice to fail

Vulnerability Detail

MajoraBalancerDepositBlock.sol#L94-L108

```
for (uint256 i = 0; i < tokens.length; i++) {
    if (address(tokens[i]) == pool) {
        continue;
    } else if (fromToken == address(0)) {
        fromToken = address(tokens[i]);
    } else {
        toToken = address(tokens[i]);
    }
}

uint256 pricePortal = IMajoraPortal(portal).getOracleRate(
    fromToken,
    toToken,
    IERC20Metadata(fromToken).decimals()
);
```

We see above that `fromToken` is always the first token and `toToken` is always the second. Balancer returns tokens sorted alphabetically. This means that depending on the token addresses, sometimes the underlying will be second and the other asset will be second. When this is the case, the portal price will inverted (i.e. wETH/wstETH instead of wstETH/wETH) and the price check will fail.

Impact

Blocks will be nonfunctional for some pools

Code Snippet

MajoraBalancerDepositBlock.sol#L94-L108

Tool used

Manual Review

Recommendation

Order should be checked and underlying should always be set as the `toToken`

Discussion

IAm0x52

Fix confirmed: 6df8e0f

Issue M-6: The onlyOwner modifier in the MajoraPortalBalancerOracleFacet contract is not functional.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/170

Summary

The onlyOwner modifier in the MajoraPortalBalancerOracleFacet contract is not functional.

Vulnerability Detail

https://github.com/sherlock-audit/2024-11-majora/blob/c057daa73301a00d13b5e67141e8cf996db152f5/portal/contracts/facets/MajoraPortalBalancerOracleFacet.sol#L29-L29

```
@>> contract MajoraPortalBalancerOracleFacet is UsingDiamondOwner {
        address public constant VAULT = 0xBA12222222228d8Ba445958a75a0704d566BF2C8;

        error InvalidInvariant();

        constructor() {}

@>>     function setBalancerWeightedMath(address _weightedMath) external onlyOwner {
            LibOracle.setBalancerWeightedMath(_weightedMath);
        }
}
```

Although the MajoraPortalBalancerOracleFacet contract inherits from UsingDiamondOwner, the AccessStorage struct has removed the owner field, meaning there is no storage space for the owner. As a result, the onlyOwner modifier reads an incorrect address for the owner data.

```
struct AccessStorage {
      IMajoraAccessManager authority;
      bool consumingSchedule;

      IMajoraAddressesProvider addressProvider;
  }
```

Impact

The setBalancerWeightedMath function is not functional. If an address happens to match the incorrect data, that address could manipulate the weightedMath.

Recommendation

Use UsingDiamondAuthority instead of UsingDiamondOwner, and replace onlyOwner with restricted.

Discussion

ZeroTrust01

fixed: e708e98

Issue M-7: The portalBridge() function uses an incorrect version of the Portal interface, rendering the function unusable.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/169

Summary

The portalBridge() function uses an incorrect version of the Portal interface, causing it to revert every time the function is called.

Vulnerability Detail

https://github.com/sherlock-audit/2024-11-majora/blob/c057daa73301a00d13b5e67141e8cf996db152f5/fees/contracts/MajoraFeeCollectorGateway.sol#L127-L127

```
function portalBridge(
        uint8 _route,
        address _approvalAddress,
        address _sourceAsset,
        address _targetAsset,
        uint256 _amount,
        uint256 _targetChain,
        bytes calldata _routeParams
    ) external restricted {
        address portal = addressProvider.portal();
        IERC20(_sourceAsset).safeIncreaseAllowance(portal, _amount);
        IMajoraPortal(portal).swapAndBridge(
            false,
@>            false,
            _route,
            _approvalAddress,
            _sourceAsset,
            _targetAsset,
            _amount,
            _targetChain,
            "",
            _routeParams
        );

    }
```

Using the old version 0.2.7 of the Portal interface, the newer version of the interface lacks the targetIsVault parameter. This will cause the call to revert.

Impact

Causing it to revert every time the function is called.

Recommendation

Update it to use the correct version of the Portal interface.

Discussion

ZeroTrust01

fixed: 57ea212

Issue M-8: The getAssetPrice() function does not check the price's timestamp for validity. Using outdated prices could lead to potential losses.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/168

Summary

The getAssetPrice() function does not check the price's timestamp for validity. Using outdated prices could lead to potential losses.

Vulnerability Detail

https://github.com/sherlock-audit/2024-11-majora/blob/c057daa73301a00d13b5e67141e8cf996db152f5/portal/contracts/libraries/LibOracle.sol#L80-L80

```
function getAssetPrice(
        address _asset
    ) internal view returns (uint256 price) {

       OracleEntry storage oracle = oracleStorage().entries[_asset];
        if(!oracle.enabled) revert OracleNotEnabled(_asset);

        if(oracle.adapterType == MajoraOracleAdaptersType.CHAINLINK) {
@>              price = uint256(AggregatorInterface(oracle.adapter).latestAnswer());
        }

        if(oracle.adapterType == MajoraOracleAdaptersType.MAJORA_ORACLE) {
@>              price = oracle.price;
        }
    }
```

Missing comparison of the updateTime for the price with the current time

An outdated price may result in losses, and Chainlink's price could also be outdated. However, MAJORA_ORACLE doesn't record the updateTime at all.

2, L2 Sequencer Uptime Feeds need to be checked for L2 chainlink oracle

Impact

Using outdated prices could lead to potential losses.

Recommendation

Add a check to compare the price’s timestamp with the current time to ensure it is up-to-date.L2 Sequencer Uptime Feeds need to be checked for L2 chainlink oracle

Discussion

ZeroTrust01

fixed: abdc5b0b

Issue M-9: MajoraGammaUniDepositBlock-/MajoraGammaQuickDepositBlock uses incorrect amount of tokens for depositing

Source: https://github.com/sherlock-audit/2024-11-majora/issues/166

Summary

Input tokens are used for both swap and LP deposit. However, the swap amount is not deducted before performing deposit.

Vulnerability Detail

The workflow is:

1. Calculate total amount of token0 to be used.
2. Swap a portion of token0 to token1.
3. Pair the token1 with the remaining of token0 for LP deposit.

The issue is the amount of token0 used in step3 still assumes the whole token0 amount in token1, and does not remove the token0 used for swap in step 2.

- https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/uni/MajoraGammaUniDepositBlock.sol#L271-L277
- https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/quick/MajoraGammaQuickDepositBlock.sol#L268-L274

```
    function enter(uint256 _index) external {
        ...

@>      uint256 amountToDeposit =
↪   (IERC20(parameters.token).balanceOf(address(this)) * parameters.tokenInPercent)
↪   /
            10000;

        ...

        IERC20(parameters.token).safeIncreaseAllowance(address(portal),
↪   params.amount);

        portal.majoraBlockSwap(
            params.route,
            params.approvalAddress,
            params.sourceAsset,
            params.targetAsset,
            params.amount,
            params.data
        );
        ...

        if (parameters.token == parameters.token0) {
            uint256 token0In;

            (uint256 amountStartT, uint256 amountEndT) = uniProxy.getDepositAmount(
                address(parameters.hypervisor),
                parameters.token1,
                amountTokenTargetToDeposit
            );

            if (amountToDeposit > amountEndT) {
                token0In = amountEndT - 1;
            } else if (amountToDeposit >= amountStartT) {
@>                  token0In = amountToDeposit;
            } else {
                revert IncorrectRatio();
            }


    ↪   IERC20(parameters.token0).safeIncreaseAllowance(address(parameters.hypervisor),
    ↪   token0In);

    ↪   IERC20(parameters.token1).safeIncreaseAllowance(address(parameters.hypervisor),
    ↪   amountTokenTargetToDeposit);

            uniProxy.deposit(token0In, amountTokenTargetToDeposit, address(this),
 ↪   parameters.hypervisor, minIn);
        }
        ...
    }
```

Impact

The block execution would fail.

Recommendation

Remove the amount of tokens used for swap: `amountToDeposit -= params.amount`.

Discussion

pkqs90

@bliiitz This is only fixed for MajoraGammaUniDepositBlock but not for MajoraGammaQuickDepositBlock.

https://github.com/majora-finance/gamma-blocks/pull/1/files

pkqs90

Fix confirmed: https://github.com/majora-finance/gamma-blocks/pull/1/commits/e32bd6105888f1d3eb3140b27a35f50c4df1d8dc

Issue M-10: MajoraAaveV3BorrowPosition Manager should use desired health factor for validation when calculating borrow amount

Source: https://github.com/sherlock-audit/2024-11-majora/issues/160

Summary

When we are performing a borrow action, MajoraAaveV3BorrowPositionManager is responsible for calculating the amount of tokens to borrow so we can hit the desired health factor. However, the precheck incorrectly uses `healthfactor.max` instead of `healthfactor.desired`, so nothing would be borrowed even if currently is larger than desired health factor.

Vulnerability Detail

https://github.com/sherlock-audit/2024-11-majora/blob/main/borrow-module/contracts/aave/MajoraAaveV3BorrowPositionManager.sol#L354

```
    function _getBorrowAmountToMatchHealthfactor() internal view returns (uint256) {
        /**
         * Fetch Aave user account data
         */
        (
            uint256 totalCollateralBase,
            uint256 totalDebtBase,
            ,
            uint256 currentLiquidationThreshold,
            ,
            uint256 healthFactor
        ) = _getUserAccountData(address(this));

        /**
         * If health factor match with parameters: skip
         */
@>      if (healthFactor <= _position.healthfactor.max) {
            return 0;
        }
        ...
    }
```

Impact

Borrow amount is incorrect.

Recommendation

Change the check to `healthfactor.desired`.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/borrow-module/commit/a44b652c1fd994983f7f82dd0962f6217d5c8056

Issue M-11: MajoraAaveV3BorrowPositionManagerDataAggregator uses incorrect OracleState during rebalance repay scenario.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/159

Summary

MajoraAaveV3BorrowPositionManagerDataAggregator is used for calculating parameters required for position manager rebalance. However, for exit strategies (repay scenario), the parameters are incorrectly calculated.

Vulnerability Detail

We can see that `dataAggregator.getPartialVaultStrategyExitExecutionInfo` is called to calculate the initial OracleState to be used for repaying debt. The issue is that we use the OracleState `_from[0]`, when it actually should be `_from[_from.length - 1]`, because the for-loop inside `getPartialVaultStrategyExitExecutionInfo()` goes upwards.

- https://github.com/sherlock-audit/2024-11-majora/blob/main/borrow-module/contracts/aave/MajoraAaveV3BorrowPositionManagerDataAggregator.sol#L92

```
    info.partialExit = dataAggregator.getPartialVaultStrategyExitExecutionInfo(
        info.vault, _from, _to
    );
    ...

    DataTypes.OracleState memory oracleState;
    if(info.partialExit.blocksInfo.length > 0)
@>      oracleState = info.partialExit.blocksInfo[_from[0]].oracleStatus;
    else
        oracleState = info.partialExit.startOracleStatus;
```

- https://github.com/majora-finance/core/blob/develop/contracts/MajoraDataAggregator.sol#L250

```
    function getPartialVaultStrategyExitExecutionInfo(address _vault, uint256[]
 ↪  memory _from, uint256[] memory _to)
        public
        view
        returns (DataTypes.MajoraVaultExecutionInfo memory info)
    {
        ...
@>      for (uint256 i = 0; i < _from.length; i++) {
```

```
            for (uint256 j = _to[i]; j >= _from[i]; j--) {
                ...
            }
        }
    }
```

Impact

MajoraAaveV3BorrowPositionManagerDataAggregator will fail to calculate rebalance parameters for the repay scenario.

Recommendation

Use `_from[_from.length - 1]` instead.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/borrow-module/commit/56de9a0e3d821faac806e9049ab8f373b2c399b5

Issue M-12: Leftover debtTokens will remain in MajoraAaveV3BorrowPositionManager during repay.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/158

Summary

When MajoraAaveV3BorrowBlock performs a repay action, if the amount of debtToken is not enough, the following logic is executed.

1. Repay a portion of the debt
2. Withdraw collateral
3. Swap collateral to debtToken
4. Finish repay

The issue here is, the swap in step 3 may swap out more debtTokens than required, and the leftover debtTokens are not refunded back to the vault.

Vulnerability Detail

Because the swap is precalculated in another transaction by `dynamicParamsInfo()`, the swap paramters may not be fully accurate when we execute it, so there may be leftover debtTokens.

- https://github.com/sherlock-audit/2024-11-majora/blob/main/borrow-module/contracts/aave/MajoraAaveV3BorrowPositionManager.sol#L195-L213

```
if (amountToRepay > debtTokenBalance) {
    if (_dynamicParams.length > 0) {
        DataTypes.DynamicSwapData memory params = abi.decode(_dynamicParams,
↪   (DataTypes.DynamicSwapData));
        _repay(address(_position.debt.token), debtTokenBalance,
↪   _position.debt.debtType, address(this));
        amountToRepay -= debtTokenBalance;

        _withdraw(address(collateral), params.amount, address(this));
        _swap(params);

        dynParamsUsed = true;
        uint256 newDebtTokenBalance = _position.debt.token.balanceOf(address(this));

        if (amountToRepay > newDebtTokenBalance) {
            revert RepayNotCoveredWithSwap();
        }
    } else {
        revert DynamicParametersNeeded();
    }
}
```

Impact

debtTokens may be leftover in the position manager, instead of refunding it to the vault.

Recommendation

Refund the leftover debtTokens.

Discussion

IAm0x52

Fix confirmed: dc5a823

Issue M-13: MajoraGammaUniDepositBlock-/MajoraGammaQuickDepositBlock does not take Gamma protocol fees into account.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/157

Summary

In Gamma hypervisor (a UniV2-like AMM pool), there is a 20% protocol fee that is taken from the regular LP fees. However, this is not accounted for in Majora blocks.

Vulnerability Detail

See the Gamma protocol fee logic here: https://github.com/GammaStrategies/hypervisor/blob/master/contracts/Hypervisor.sol#L166-L176

```
    function _zeroBurn(int24 tickLower, int24 tickUpper) internal returns(uint128
 ↪  liquidity) {
      /// update fees for inclusion
      (liquidity, ,) = _position(tickLower, tickUpper);
      if(liquidity > 0) {
        pool.burn(tickLower, tickUpper, 0);
        (uint256 owed0, uint256 owed1) = pool.collect(address(this), tickLower,
 ↪  tickUpper, type(uint128).max, type(uint128).max);
        emit ZeroBurn(fee, owed0, owed1);
@>      if (owed0.div(fee) > 0 && token0.balanceOf(address(this)) > 0)
 ↪  token0.safeTransfer(feeRecipient, owed0.div(fee));
@>      if (owed1.div(fee) > 0 && token1.balanceOf(address(this)) > 0)
 ↪  token1.safeTransfer(feeRecipient, owed1.div(fee));
      }
    }
```

For a live contract example, the `fee` is set to `5`, which is a 20% protocol fee.

There are multiple places in Majora blocks that use the pending LP fees without deducting Gamma protocol fees:

1. https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/quick/MajoraGammaQuickDepositBlock.sol#L194-L195
2. https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/quick/MajoraGammaQuickDepositBlock.sol#L360-L361
3. https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/quick/MajoraGammaQuickDepositBlock.sol#L535-L536

Impact

Estimations of `dynamicParamsInfo()`, and `oracleEnter/oracleExit` would be inaccurate.

Recommendation

Deduct Gamma protocol fees accordingly.

Discussion

pkqs90

Fix confirmed.

Issue M-14: Potential integer overflow in MajoraGammaUniDepositBlock/ MajoraGammaQuickDepositBlock during price calculation.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/156

Summary

In the two blocks `MajoraGammaUniDepositBlock/MajoraGammaQuickDepositBlock`, uniswap price calculation is used. There is a potential integer overflow issue, which would cause DoS.

Vulnerability Detail

There are four occurences of this issue:

- https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/uni/MajoraGammaUniDepositBlock.sol#L81
- https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/uni/MajoraGammaUniDepositBlock.sol#L347
- https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/quick/MajoraGammaQuickDepositBlock.sol#L76
- https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/quick/MajoraGammaQuickDepositBlock.sol#L351

The issue arises by performing a `sqrtPriceX96 * sqrtPriceX96`. `sqrtPriceX96` is a uint160, and multiplying itself may overflow uint256.

```
        (, int24 currentTick, , , , , ) = IAlgebraPool(pool).globalState();
        uint160 sqrtPriceX96 = TickMath.getSqrtRatioAtTick(currentTick);
@>      uint256 spotPrice = FullMath.mulDiv(uint256(sqrtPriceX96) *
 ↪  uint256(sqrtPriceX96), unitA, 2 ** (96 * 2));
```

Impact

Potential DoS during block execution.

Recommendation

Similar to how UniV3 handles it: https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/OracleLibrary.sol#L64

```
if (sqrtRatioX96 <= type(uint128).max) {
    uint256 ratioX192 = uint256(sqrtRatioX96) * sqrtRatioX96;
    quoteAmount = baseToken < quoteToken
        ? FullMath.mulDiv(ratioX192, baseAmount, 1 << 192)
        : FullMath.mulDiv(1 << 192, baseAmount, ratioX192);
} else {
    uint256 ratioX128 = FullMath.mulDiv(sqrtRatioX96, sqrtRatioX96, 1 << 64);
    quoteAmount = baseToken < quoteToken
        ? FullMath.mulDiv(ratioX128, baseAmount, 1 << 128)
        : FullMath.mulDiv(1 << 128, baseAmount, ratioX128);
}
```

Discussion

pkqs90

Added comment in https://github.com/majora-finance/gamma-blocks/pull/1/files.

Issue M-15: The boundary value for the minimum HealthFactor in MajoraAaveV3PositionManagerCommons is incorrect.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/154

Summary

The boundary value for the minimum HealthFactor in MajoraAaveV3PositionManagerCommons is incorrect, which could potentially lead to the vault being liquidated.

Vulnerability Detail

```
function _verifyHealthfactor(uint256 _min, uint256 _desired, uint256 _max) internal
 ↪  pure {
@>        if(_min <= 1 || _max <= _min || _desired <= _min || _desired >= _max)
 ↪  revert BadHealthfactorConfiguration();
    }
```

totalDebtBaseDesired = (totalCollateralInBaseCurrency * currentLiquidationThreshold) / 10000) / hfDesired

maxBorrow = (totalCollateralInBaseCurrency * maxLTV) / 10000)

totalDebtBaseDesired<=maxBorrow Then get: currentLiquidationThreshold/hfDesired <= maxLTV hfDesired>=currentLiquidationThreshold/maxLTV So _min = 1e18*currentLiquidationThreshold/maxLTV

Impact

Since _min is set by the vault owner, improper configuration by the owner, combined with the lack of proper checks in the function, could result in the vault being liquidated.

Recommendation

if _min <= 1e18 * currentLiquidationThreshold / maxLTV, revert BadHealthfactorConfiguration()

Discussion

ZeroTrust01

fixed: ed160a0

Issue M-16: In MajoraCurveDepositBlock, when a token's index in the Curve pool is 0, it can result in a DoS (Denial of Service).

Source: https://github.com/sherlock-audit/2024-11-majora/issues/153

Summary

When the token index is 0, 0 - 1 error occurs, causing the transaction to revert().

Vulnerability Detail

```
function _checkprice(BlockParameters memory parameters) internal view {
        if (parameters.poolType == TypePool.CurveStableSwapNG) {
            /* CHECK PRICE */
            //Function to calculate the exponential moving average (EMA) price for
 ↪  the coin at index i with regard to the coin at index 0.
@>            uint256 emaPrice =
 ↪  ICurveStableSwapNG(parameters.pool).price_oracle(parameters.index - 1);
@>            uint256 currentPrice =
 ↪  ICurveStableSwapNG(parameters.pool).get_p(parameters.index - 1);

           //..........
    }
```

When the token index is 0, 0 - 1 error occurs, causing the transaction to revert().

Impact

This causes the core enter() and exit() functions to revert.

Recommendation

Add special handling for cases where the token index is 0.

Discussion

ZeroTrust01

fixed: e63c029

Issue M-17: The debt token ratio calculation in the MajoraAaveV3BorrowBlock:: oracleExit() function is incorrect.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/152

Summary

The debt token ratio calculation in the MajoraAaveV3BorrowBlock::oracleExit() function is incorrect, which may result in an inaccurate calculation of the vault's TVL (Total Value Locked).

Vulnerability Detail

```
function oracleExit(
        DataTypes.OracleState memory _before,
        bytes memory _parameters,
        uint256 _percent
    ) external view returns (DataTypes.OracleState memory) {
      //......
        uint256 collateralReturned = IMajoraAaveV3BorrowPositionManager(
            parameters.positionManager
@>>        ).returnedAmountAfterRepay(debtTokenBal, _percent);

        oracleState.removeTokenAmount(parameters.debt, debtTokenBal * _percent /
↪   10000);
        oracleState.addTokenAmount(parameters.collateral, collateralReturned);
        return oracleState;
    }
}
```

```
function returnedAmountAfterRepay(uint256 _repayAmount, uint256 _percent) external
↪   view returns (uint256) {
        address[] memory assets = new address[](2);
        assets[0] = address(_position.collateral.token);
        assets[1] = address(_position.debt.token);
        uint256[] memory prices = oracle.getAssetsPrices(assets);

        uint256 collateralBaseReturned;
@>>     uint256 repayBase = (_repayAmount * prices[1]) / 10 **
↪   _position.debt.decimals;
        (uint256 totalCollateralBase, uint256 totalDebtBase, , , , ) =
↪   _getUserAccountData(address(this));
```

```
        uint256 collateralBase = (totalCollateralBase * _percent) / 10000;
        uint256 debtBase = (totalDebtBase * _percent) / 10000;

        if (repayBase >= debtBase) {
            collateralBaseReturned = collateralBase + (repayBase - debtBase);
        } else {
            collateralBaseReturned = collateralBase - (debtBase - repayBase);
        }

        uint256 collateralAmount = (collateralBaseReturned * 10 **
↪   _position.collateral.decimals) / prices[0];
        return collateralAmount;
    }
```

In returnedAmountAfterRepay(), is repay the whole debtTokenBal. But in oracleExit only remove debtTokenBal * _percent / 10000

Impact

may result in an inaccurate calculation of the vault's TVL (Total Value Locked).

Recommendation

Modify the debtToken to use the correct ratio for accurate calculations.

Discussion

ZeroTrust01

fixed: 5dd1562

Issue M-18: The calculation error in Majora-BalancerDepositBlock::_checkPrice().

Source: https://github.com/sherlock-audit/2024-11-majora/issues/151

Summary

The _checkPrice() function does not account for token precision and incorrectly calculates the fromAmount, resulting in the _checkPrice() function losing its price protection effectiveness.

Vulnerability Detail

https://github.com/sherlock-audit/2024-11-majora/blob/2f77e1e2b37507bc7d044e1d44fdf6898d063538/balancer-v2-blocks/contracts/MajoraBalancerDepositBlock.sol#L81

```
    function _checkPrice(BlockParameters memory parameters) private view returns
 ↪  (bool success) {
        if (parameters.poolType == PoolType.COMPOSABLE_STABLE) {
            (address pool, ) = vault.getPool(parameters.poolId);

@>>             uint256 rate = IComposableStablePool(pool).getRate();

            (IERC20[] memory tokens, , ) = vault.getPoolTokens(parameters.poolId);

            require(tokens.length == 3, "Expected exactly 3 tokens");

            address fromToken;
            address toToken;

            for (uint256 i = 0; i < tokens.length; i++) {
                if (address(tokens[i]) == pool) {
                    continue;
                } else if (fromToken == address(0)) {
                    fromToken = address(tokens[i]);
                } else {
                    toToken = address(tokens[i]);
                }
            }

            uint256 pricePortal = IMajoraPortal(portal).getOracleRate(
                fromToken,
                toToken,
@>>                10 ** IERC20Metadata(fromToken).decimals()
            );
        //....
    }
```

1. The function IComposableStablePool(pool).getRate() returns the appreciation of BPT relative to the underlying tokens, as an 18 decimal fixed point number. So there are two issues that need to be rechecked here: 1. This is not the to/from price. Do you want to use this price to replace the to/from price? 2. The precision here is fixed at 18. If the precision of to is not 18, it cannot be directly compared with pricePortal. balancer/balancer-v2-monorepo@e16fad4#diff-9906a62f599f28724fd4b0da7e9f885696b348fefc5cd29dd463fd2de14c3627R1030
2. When calling getOracleRate(), IERC20Metadata(fromToken).decimals() was incorrectly used.

Impact

The _checkPrice() function either loses its price protection effectiveness or, due to calculation errors, results in significant price discrepancies, leading to a potential DoS (Denial of Service).

Recommendation

```
1.  Reimplement the price calculation for toToken and fromToken.
2.  Use 10**IERC20Metadata(fromToken).decimals() instead of
 ↪  IERC20Metadata(fromToken).decimals().
```

Discussion

ZeroTrust01

fixed: 8ff1395

Issue M-19: BalancerDepositBlock does not check price when minting/redeeming BPT, and is vulnerable to frontrunning.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/146

Summary

The `_checkPrice()` function is implemented to make sure frontrunning price manipulation doesn't happen when minting/redeeming BPT. This function returns a boolean value. However, the result of this function is not used, hence the price check does not actually work.

Vulnerability Detail

Note that the price check is currently the only mechanism to avoid frontrunning, because slippage parameter is set to 0.

- https://github.com/sherlock-audit/2024-11-majora/blob/main/balancer-v2-blocks/contracts/MajoraBalancerDepositBlock.sol#L81

```
    function _checkPrice(BlockParameters memory parameters) private view returns
 ↪  (bool success) {
        ...
    }

    function enter(uint256 _index) external whenNotInVaultContext {
        BlockParameters memory parameters =
 ↪  abi.decode(LibBlock.getStrategyStorageByIndex(_index), (BlockParameters));
@>      _checkPrice(parameters);
        ...
    }

    function exit(uint256 _index, uint256 _percent) external whenNotInVaultContext {
        BlockParameters memory parameters =
 ↪  abi.decode(LibBlock.getStrategyStorageByIndex(_index), (BlockParameters));
@>      _checkPrice(parameters);
        ...
    }
```

Impact

Minting/redeeming BPT tokens may be frontrun, and users will receive less tokens than expected.

Recommendation

Change to `require(_checkPrice(parameters));`.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/balancer-v2-blocks/pull/1/commits/8ff139593c70c40f03186508212d799571e971b2

Issue M-20: `stakersCooldowns[]` is not correctly updated when users are staking MAJ token.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/145

Summary

When users are staking MAJ token, the cooldown is updated twice, leading to a longer cooldown period than expected.

Vulnerability Detail

In the `stake()` function, the `stakersCooldowns[onBehalfOf]` function is updated first, and it is updated again in the internal `_update()` ERC20 function which is called during minting tokens.

Since we use the current timestamp as cooldown time for newly minted tokens, the final cooldown time would be larger than expected, because it is updated twice.

https://github.com/sherlock-audit/2024-11-majora/blob/main/fees/contracts/stkMAJ/StkMAJ.sol#L47

```
    function stake(address onBehalfOf, uint256 amount) external override {
        if(amount == 0) revert InvalidZeroAmount();

        uint256 balanceOfUser = balanceOf(onBehalfOf);
@>      stakersCooldowns[onBehalfOf] = getNextCooldownTimestamp(0, amount,
 ↪  onBehalfOf, balanceOfUser);

        _mint(onBehalfOf, amount);
        IERC20(token).safeTransferFrom(msg.sender, address(this), amount);

        emit Staked(msg.sender, onBehalfOf, amount);
    }

    // The following functions are overrides required by Solidity.
    function _update(address from, address to, uint256 amount)
        internal
        override
    {
        if (from != to) {
            uint256 balanceOfFrom = balanceOf(from);
            uint256 balanceOfTo = balanceOf(to);
            uint256 previousSenderCooldown = stakersCooldowns[from];
@>          stakersCooldowns[to] = getNextCooldownTimestamp(previousSenderCooldown,
↪   amount, to, balanceOfTo);
            // if cooldown was set and whole balance of sender was transferred -
↪   clear cooldown
            if (balanceOfFrom == amount && previousSenderCooldown != 0) {
                stakersCooldowns[from] = 0;
            }
        }

        super._update(from, to, amount);
    }
```

Impact

Cooldown periods will be longer than expected when staking MAJ tokens.

Recommendation

Only update the cooldown period once.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/fees/pull/1/commits/83b47520017488813a872e1768cd84cf417fcba8

Issue M-21: MajoraMerklHarvestBlock does not use correct interface for claiming rewards.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/144

Summary

MajoraMerklHarvestBlock does not use correct interface for claiming rewards.

Vulnerability Detail

This block is used for harvesting rewards on a merkl distributor: https://polygonscan.com/address/0x3Ef3D8bA38EBe18DB133cEc108f4D14CE00Dd9Ae.

However, the interfaces do not match. The live contract spreads out the parameters, while the one in MajoraMerklHarvestBlock packs them in a struct.

```
function claim(
     address[] calldata users,
     address[] calldata tokens,
     uint256[] calldata amounts,
     bytes32[][] calldata proofs
 ) external {
     ...
 }
```

https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/MajoraMerklHarvestBlock.sol#L46

```
    interface IMerklClaim {
        struct ClaimParams {
            address[] users;
            address[] tokens;
            uint256[] amounts;
            bytes32[][] proofs;
        }

        function claim(ClaimParams calldata params) external;
    }

    function harvest(uint256 _index) external {
        bytes memory dynParameters = LibBlock.getDynamicBlockData(_index);
        if (dynParameters.length > 0) {
            IMerklClaim.ClaimParams memory claimParams = abi.decode(dynParameters,
 ↪  (IMerklClaim.ClaimParams));

@>          IMerklClaim(merklDistributor).claim(claimParams);
        }
    }
```

Impact

Rewards cannot be claimed in MajoraMerklHarvestBlock.

Recommendation

Implement the correct interface.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/gamma-blocks/pull/1/files

Issue M-22: MajoraOperationsPaymentToken may be vulnerable to gas grief attack when removing approvals.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/143

Summary

Malicious users can DoS users from removing approval to a spender by conducting a gas grief attack.

Vulnerability Detail

Considering that anyone can approve an allowance to spender (vault), a grief attack may be performed by a malicious user by creating a lot of addresses and approving 1 wei to the vault. This way when honest users (sponsors) want to remove an approval, they may have to go through the entire sponsor list to remove their address, which would end up in a lot of gas, or even out-of-gas DOS.

- https://github.com/sherlock-audit/2024-11-majora/blob/main/mopt/contracts/MajoraOperationsPaymentToken.sol#L214

```
function _removeSponsor(address _sponsor, address _spender) internal {
    uint256 sponsorLength = sponsors[_spender].length;
    for (uint256 i = 0; i < sponsorLength; i++) {
        if (sponsors[_spender][i] == _sponsor) {
            sponsors[_spender][i] = sponsors[_spender][sponsorLength - 1];
            sponsors[_spender].pop();
            isSponsor[_spender][_sponsor] = false;
            return;
        }
    }
}
```

Simple poc. Add the following code in `mopt/test/main.ts`. For N=100, the gas spent is ~3e5. For N=200, gas is ~6e5. So for the worst case, the gas limit 30M may be exceeded when N is around 10000.

```
it("PoC. Out-of-gas when removing sponsor.", async function () {

  const { randomBytes, hexlify } = require("ethers/lib/utils");
  const contract = <MajoraOperationsPaymentToken>(
    this.MajoraOperationsPaymentToken
  );
  const [deployer, vault] = await ethers.getSigners();
  const N = 100;
  let lastUser = "";

  // Deploy wallets and each approve vault for 1e17 MOPT.
  for (let i = 0; i < N; i++) {
    const user = new ethers.Wallet(hexlify(randomBytes(32)), ethers.provider);
    if (i == N-1) {
      lastUser = user;
    }
    await deployer.sendTransaction({
      to: user.address,
      value: ethers.utils.parseEther("0.2"),
    });
    await contract.connect(user)["mint()"]({ value: ethers.utils.parseEther("0.1")
↪   });
    await contract.connect(user).approveOperation(vault.address,
↪   ethers.utils.parseEther("0.1"));
  }

  const tx = await contract.connect(lastUser).approveOperation(vault.address, 0);
  const receipt = await tx.wait();
  console.log(receipt);
});
```

Impact

Users may fail to remove approval to a a spender due to gas griefing attack.

Recommendation

Consider using the EnumberableSet from openzeppelin instead https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/structs/EnumerableSet.sol#L232, to reduce O(n) complexity to O(1).

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/mopt/pull/1/commits/df4ba39519cbe050cf6152e387c38bc3da26f110

Issue M-23: MajoraStargateStakeLPBlock-/MajoraStargateDepositAndStakeLPBlock `oracleExit()` does not include rewards tokens.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/141

Summary

The Stargate staking contract always withdraw all rewards tokens when LP token is withdrawn. However, the reward tokens are not accounted for in oracleExit function.

Vulnerability Detail

In Stargate staking contract, the `withUpdate` variable is always set to true when we are withdrawing LP tokens.

https://github.com/sherlock-audit/2024-11-majora/blob/main/stargate-blocks/contracts/MajoraStargateStakeLPBlock.sol#L94

```
    function exit(uint256 _index, uint256 _percent) external {
        BlockParameters memory parameters =
↪   abi.decode(LibBlock.getStrategyStorageByIndex(_index), (BlockParameters));
        uint256 deposited = lpStaking.balanceOf(IERC20(parameters.token),
↪   address(this));

        uint256 amountToUnstake = (deposited * _percent) / 10000;

        lpStaking.withdraw(IERC20(parameters.token), amountToUnstake);
    }

@>  function oracleExit(
        DataTypes.OracleState memory _before,
        bytes memory _parameters,
        uint256 _percent
    ) external view returns (DataTypes.OracleState memory) {
        BlockParameters memory parameters = abi.decode(_parameters,
↪   (BlockParameters));
        DataTypes.OracleState memory oracleState = _before;

        uint256 amountDeposited = lpStaking.balanceOf(IERC20(parameters.token),
↪   oracleState.vault);
        uint256 amountToWithdraw = (amountDeposited * _percent) / 10000;
        oracleState.addTokenAmount(parameters.token, amountToWithdraw);
        return oracleState;
    }
```

https://github.com/stargate-protocol/stargate-v2/blob/main/packages/stg-evm-v2/src/peripheral/rewarder/StargateStaking.sol#L60

```
    function withdraw(IERC20 token, uint256 amount) external override nonReentrant
 ↪  validPool(token) {
@>      _pools[token].withdraw(token, msg.sender, msg.sender, amount, true);
    }
```

https://github.com/stargate-protocol/stargate-v2/blob/main/packages/stg-evm-v2/src/peripheral/rewarder/lib/StakingLib.sol#L45

```
    function withdraw(
        StakingPool storage self,
        IERC20 token,
        address from,
        address to,
        uint256 amount,
        bool withUpdate
    ) internal {
        uint256 oldBal = self.balanceOf[from];
        uint256 oldSupply = self.totalSupply;

        if (oldBal < amount) revert WithdrawalAmountExceedsBalance();

        uint256 newBal = oldBal - amount;

        self.balanceOf[from] = newBal;
        self.totalSupply = oldSupply - amount;

        emit IStargateStaking.Withdraw(token, from, to, amount, withUpdate);

@>      if (withUpdate) {
            self.rewarder.onUpdate(token, from, oldBal, oldSupply, newBal);
        }
        token.safeTransfer(to, amount);
    }
```

Impact

OracleState is not correctly calculated.

Recommendation

Add the rewards in OracleExit functions.

Discussion

pkqs90

Added comment in https://github.com/majora-finance/stargate-blocks/pull/1: Reward should be fully updated instead of only updating a percentage.

pkqs90

Fix confirmed: https://github.com/majora-finance/stargate-blocks/pull/1/commits/b0decadc92210b127a092fbab3c4cac241b5be57

Issue M-24: MajoraFeeCollectorGateway `portalBridge()` function should send native tokens as fees.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/140

Summary

MajoraFeeCollectorGateway's `portalBridge()` function is used for bridging fees to the main chain. Most bridging services require bridge fees in native token, so this function should be payable, and native token should be passed along to portal for bridging.

Vulnerability Detail

```
    function portalBridge(
        uint8 _route,
        address _approvalAddress,
        address _sourceAsset,
        address _targetAsset,
        uint256 _amount,
        uint256 _targetChain,
        bytes calldata _routeParams
@>  ) external restricted {
        address portal = addressProvider.portal();
        IERC20(_sourceAsset).safeIncreaseAllowance(portal, _amount);
@>      IMajoraPortal(portal).swapAndBridge(
            false,
            false,
            _route,
            _approvalAddress,
            _sourceAsset,
            _targetAsset,
            _amount,
            _targetChain,
            "",
            _routeParams
        );
        ...
    }
```

- https://github.com/sherlock-audit/2024-11-majora/blob/main/fees/contracts/MajoraFeeCollectorGateway.sol#L82

Impact

Bridging would fail if native tokens are required as fees.

Recommendation

1. Make the function payable;
2. Pass the native tokens when calling `IMajoraPortal(portal).swapAndBridge()`.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/fees/pull/1/commits/a05485517d5f7e11a348997a7eb426143f505df5

Issue M-25: MajoraCurveHarvestBlock `harvest()` function should also mint CRV rewards.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/139

Summary

MajoraCurveHarvestBlock's `harvest()` function only claims extra rewards, but not CRV rewards.

Vulnerability Detail

`IGauge(parameters.gauge).factory().mint(parameters.gauge)` should be called to claim CRV rewards.

- https://docs.curve.fi/liquidity-gauges-and-minting-crv/xchain-gauges/ChildGaugeFactory/#mint
- https://github.com/curvefi/curve-xchain-factory/blob/master/contracts/ChildGaugeFactory.vy#L147

Impact

CRV rewards would be left unclaimed.

Recommendation

Add `IGauge(parameters.gauge).factory().mint(parameters.gauge)` in `harvest()` function.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/curve-blocks/pull/1/files

Issue M-26: MajoraCurveDepositBlock `oracleEnter()` should add LP token address instead of pool address to OracleState.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/138

Summary

MajoraCurveDepositBlock handles the logic for minting LP tokens in Curve pools. There are three type of pools that the block supports.

1. CurveStableSwapNG
2. CryptoV2
3. CryptoV2Zap4Tokens

For the first pool, the pool and lp token uses the same address. However, for the latter two, they are different. In `oracleEnter()` function, the pool address is incorrectly added to OracleState instead of LP token address.

Vulnerability Detail

`parameters.lp` should be used instead of `parameters.pool`.

- https://github.com/sherlock-audit/2024-11-majora/blob/main/curve-blocks/contracts/MajoraCurveDepositBlock.sol#L223
- https://github.com/sherlock-audit/2024-11-majora/blob/main/curve-blocks/contracts/MajoraCurveDepositBlock.sol#L233

```
    function oracleEnter(
        DataTypes.OracleState memory _before,
        bytes memory _parameters
    ) external view returns (DataTypes.OracleState memory) {
        ...
        if (parameters.poolType == TypePool.CryptoV2Zap4Tokens) {
            if (parameters.zap != address(0)) {
                uint256[4] memory amounts;
                amounts[parameters.index] = amountToDeposit;
                uint256 estimatedLPTokens =
 ↪  ICurveZap4Tokens(parameters.zap).calc_token_amount(amounts);
@>              oracleState.addTokenAmount(parameters.pool, estimatedLPTokens);
                oracleState.removeTokenAmount(parameters.token, amountToDeposit);
                return oracleState;
            } else {
                revert NoZap();
            }
        }
        if (parameters.poolType == TypePool.CryptoV2) {
            uint256[2] memory amounts;
            amounts[parameters.index] = amountToDeposit;
            uint256 estimatedLPTokens =
 ↪  ICurveCryptoV2(parameters.pool).calc_token_amount(amounts);
@>          oracleState.addTokenAmount(parameters.pool, estimatedLPTokens);
            oracleState.removeTokenAmount(parameters.token, amountToDeposit);
            return oracleState;
        }
        return oracleState;
    }
```

Impact

OracleState is incorrectly calculated during strategy simulation.

Recommendation

`parameters.lp` should be used instead of `parameters.pool`.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/curve-blocks/pull/1/files

Issue M-27: MajoraBalancerHarvestBlock does not account for extra rewards other than BAL.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/137

Summary

MajoraBalancerHarvestBlock is responsible for harvesting rewards from Balancer gauge. There are two types of rewards: 1) BAL; 2) extra rewards. Only BAL token is claimed and accounted for in this block, other reward tokens will be left behind.

Vulnerability Detail

Following functions should be used to claim extra rewards:

- https://github.com/balancer/balancer-v2-monorepo/blob/master/pkg/liquidity-mining/contracts/gauges/ChildChainGauge.vy#L604
- https://github.com/balancer/balancer-v2-monorepo/blob/master/pkg/liquidity-mining/contracts/gauges/ChildChainGauge.vy#L572

```
@view
@external
def claimable_reward(_user: address, _reward_token: address) -> uint256:
    """
    @notice Get the number of claimable reward tokens for a user
    @param _user Account to get reward amount for
    @param _reward_token Token to get reward amount for
    @return uint256 Claimable reward token amount
    """
    ...


@external
@nonreentrant('lock')
def claim_rewards(
    _addr: address = msg.sender,
    _receiver: address = ZERO_ADDRESS,
    _reward_indexes: DynArray[uint256, MAX_REWARDS] = []
):
    """
    @notice Claim available reward tokens for `_addr`
    @param _addr Address to claim for
    @param _receiver Address to transfer rewards to - if set to
```

```
                    ZERO_ADDRESS, uses the default reward receiver
                    for the caller
    @param _reward_indexes Array with indexes of the rewards to be checkpointed
↪   (all of them by default)
    """
    ...
```

Impact

Rewards may be left behind for MajoraBalancerHarvestBlock.

Recommendation

Add the functions in `MajoraBalancerHarvestBlock#harvest()` and `MajoraBalancerHarvestBlock#oracleHarvest()`.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/balancer-v2-blocks/pull/1/files

Issue M-28: OracleState does not handle discovering unexisting tokens correctly in `oracleEnter/oracleExit/dynamicParamsInfo` functions.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/136

Summary

For strategy blocks, the `oracleEnter/oracleExit/dynamicParamsInfo` functions are used to simulate the strategy execution. OracleState is used to keep tract of the tokens inside the vault. During the initialization of a new token in OracleState, the entire token balance should be added in OracleState. However, this is not the case for most of the blocks.

Vulnerability Detail

There is a `percentage` concept in `oracleEnter/oracleExit/dynamicParamsInfo`, determining the amount of tokens used for this block. When adding an unexisting token, the entire balance should be added, and the used percentage should be removed.

Example: If vault has 100 tokenA, percentage = 70%. We should first add 100 tokenA into oracleState, then remove 70. We should end up with 30 tokenA left.

However, for a lot of the vaults, this is not the case. An example is the MajoraCurveGaugeBlock, which doesn't add the whole token balance.

```
    function oracleEnter(
        DataTypes.OracleState memory _before,
        bytes memory _parameters
    ) external view returns (DataTypes.OracleState memory) {
        BlockParameters memory parameters = abi.decode(_parameters,
↪   (BlockParameters));
        DataTypes.OracleState memory oracleState = _before;
        uint256 amountToDeposit = (oracleState.findTokenAmount(parameters.token) *
↪   parameters.tokenInPercent) / 10000;

        if (amountToDeposit == 0) {
@>          amountToDeposit =
                (IERC20(parameters.token).balanceOf(oracleState.vault) *
↪   parameters.tokenInPercent) /
                10000;
            oracleState.addTokenAmount(parameters.token, amountToDeposit);
        }
@>      oracleState.removeTokenAmount(parameters.token, amountToDeposit);
        oracleState.addTokenAmount(parameters.gauge, amountToDeposit);
        return oracleState;
    }

    function oracleExit(
        DataTypes.OracleState memory _before,
        bytes memory _parameters,
        uint256 _percent
    ) external view returns (DataTypes.OracleState memory) {
        BlockParameters memory parameters = abi.decode(_parameters,
 ↪  (BlockParameters));
        DataTypes.OracleState memory oracleState = _before;
        uint256 amountToWithdraw = (oracleState.findTokenAmount(parameters.gauge) *
 ↪  _percent) / 10000;

        if (amountToWithdraw == 0) {
@>          amountToWithdraw =
 ↪  (IERC20(parameters.gauge).balanceOf(oracleState.vault) * _percent) / 10000;
            oracleState.addTokenAmount(parameters.gauge, amountToWithdraw);
        }

@>      oracleState.removeTokenAmount(parameters.gauge, amountToWithdraw);
        oracleState.addTokenAmount(parameters.token, amountToWithdraw);
        return oracleState;
    }
```

For `dynamicParamsInfo()` function, it is the same case. MajoraAaveV3BorrowBlock can be viewed as an example:

```
    function dynamicParamsInfo(
        DataTypes.BlockExecutionType _exec,
        bytes memory _params,
        DataTypes.OracleState memory _oracleData,
        uint256 _percent
    ) external view returns (bool, DataTypes.DynamicParamsType, bytes memory) {
        ...
        uint256 debtTokenBal = _oracleData.findTokenAmount(parameters.debt);
        if (!_oracleData.tokenExists(parameters.debt) && debtTokenBal == 0) {
            debtTokenBal = IERC20(parameters.debt).balanceOf(_oracleData.vault) *
 ↪  _percent / 10000;
@>          _oracleData.addTokenAmount(parameters.debt, debtTokenBal);
        }
        ...
    }
```

The list of blocks that does not support this is:

For `OracleEnter/OracleExit`:

- MajoraAaveV3BorrowBlock
- MajoraAaveV3DepositBlock
- MajoraAuraDepositBlock
- MajoraBalancerDepositBlock
- MajoraBalancerGaugeBlock
- MajoraConvexDepositBlock
- MajoraCurveDepositBlock
- MajoraCurveGaugeBlock
- MajoraGammaQuickDepositBlock
- MajoraGammaUniDepositBlock
- MajoraStargateDepositAndStakeLPBlock
- MajoraStargateDepositLPBlock
- MajoraStargateStakeLPBlock

For `dynamicParamsInfo`:

- MajoraAaveV3BorrowBlock
- MajoraGammaQuickDepositBlock
- MajoraGammaUniDepositBlock
- MajoraPortalSwapBlock
- MajoraPortalSwapHarvestBlock

Impact

The result of block strategy simulation for vaults would be inaccurate.

Recommendation

If token does not exist in OracleState, add the full balance.

Discussion

IAm0x52

Commits: Aave V3: https://github.com/majora-finance/aave-v3-blocks/commit/972777fc17a0958bce7c6c640030a5b540c659cf Gamma: https://github.com/majora-finance/gamma-blocks/commit/da087efb6cf991cca89de8df0dd8d11d9463b465 Stargate:

https://github.com/majora-finance/stargate-blocks/commit/12df8ff75bf25a7dcfaa0aa8602fafbce03ee8c0 Balancer: https://github.com/majora-finance/balancer-v2-blocks/commit/554f128b40bba912ab88309dc94564621614fa2f Aura: https://github.com/majora-finance/aura-blocks/commit/a9ad80032b32bd7405714016eb409f79c1a5ff19 Curve: https://github.com/majora-finance/curve-blocks/commit/125f6f67b39c262921b4696e5e02bef551c76f78 Convex: https://github.com/majora-finance/convex-blocks/commit/15a819107bf4e8be8a72fdebb66cf47335d367fe Libraries: https://github.com/majora-finance/libraries/commit/db8f88ea8eac64e3fcfb98f29ae671cb1c423bf2

IAm0x52

Looks like all instances have been covered but would appreciate a second check

pkqs90

Also for majora blocks: https://github.com/majora-finance/majora-blocks/pull/1/commits/cf7c8fb0d0cccd7331d977cf0b9d5a1d9f5fbe07

Checked all instances. Should be good.

Issue L-1: MajoraCurveDepositBlock `exit()/oracleExit()` functions does not need to calculate `estimatedToken` as slippage.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/150

Summary

MajoraCurveDepositBlock `exit()/oracleExit()` functions redeem LP tokens from Curve pools. An `estimatedToken` is calculated and used as slippage. However, the calculation of `estimatedToken` is actually queried using the same formula as real execution, which means the slippage check would never fail.

We can remove such check to save gas.

Vulnerability Detail

- https://github.com/sherlock-audit/2024-11-majora/blob/main/curve-blocks/contracts/MajoraCurveDepositBlock.sol#L146

```
    function exit(uint256 _index, uint256 _percent) external {
        BlockParameters memory parameters =
 ↪  abi.decode(LibBlock.getStrategyStorageByIndex(_index), (BlockParameters));
        _checkprice(parameters);

        uint256 amountToWithdraw = (IERC20(parameters.lp).balanceOf(address(this))
 ↪  * _percent) / 10000;

        if (parameters.poolType == TypePool.CurveStableSwapNG) {
            IERC20(parameters.lp).safeIncreaseAllowance(parameters.pool,
 ↪  amountToWithdraw);

            //estimate output token
@>          uint256 estimatedToken =
 ↪  ICurveStableSwapNG(parameters.lp).calc_withdraw_one_coin(
                amountToWithdraw,
                parameters.index_128
            );
            ICurveStableSwapNG(parameters.lp).remove_liquidity_one_coin(
                amountToWithdraw,
                parameters.index_128,
@>              estimatedToken
            );
        }
        ...
    }
```

Impact

Gas optimization.

Recommendation

Remove `estimatedToken` and set slippage to 0 instead. Price manipulation attacks are already prevented by spot/oracle price checks.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/curve-blocks/pull/1/files

Issue L-2: MajoraConvexHarvestBlock gas usage optimization in `oracleHarvest()` function.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/149

Summary

MajoraConvexHarvestBlock's `oracleHarvest()` function queries the amount of rewards in ConvexRewardPool and adds it to OracleState. It's implementation can be optimized.

Vulnerability Detail

`IConvexRewardPool(rewards).earned(address(this))` returns an array of `EarnedData` data structure.

```
struct EarnedData {
    address token;
    uint256 amount;
}
```

The query is performed twice in each iteration of the for loop. This would make it an O(N^2) implementation. We can cache the result of `IConvexRewardPool(rewards).earned(address(this))` in the beginning to make it an O(N) implementation.

- https://github.com/sherlock-audit/2024-11-majora/blob/main/convex-blocks/contracts/MajoraConvexHarvestBlock.sol#L52

```
    function oracleHarvest(
        DataTypes.OracleState memory _previous,
        bytes memory _parameters
    ) external returns (DataTypes.OracleState memory) {
        BlockParameters memory parameters = abi.decode(
            _parameters,
            (BlockParameters)
        );
        DataTypes.OracleState memory oracleState = _previous;
        (, , address rewards, , ) = boosterLite.poolInfo(parameters.poolId);

        uint256 length = IConvexRewardPool(rewards)
            .earned(address(this))
            .length;

        for (uint256 i = 0; i < length; i++) {
```

```
@>          address token = IConvexRewardPool(rewards)
            .earned(address(this))[i].token;
@>          uint256 amount = IConvexRewardPool(rewards)
            .earned(address(this))[i].amount;
            if (amount > 0) {
                oracleState.addTokenAmount(token, amount);
            }
        }

        return oracleState;
    }
```

Impact

Gas optimization.

Recommendation

Cache the result of `IConvexRewardPool(rewards).earned(address(this))` in the beginning to make it an O(N) implementation.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/convex-blocks/pull/1/files

Issue L-3: MajoraPortalBalancerOracleFacet should use current invariant instead of last invariant when calculating LP price.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/180

Summary

MajoraPortalBalancerOracleFacet should use current invariant instead of last invariant when calculating LP price.

Vulnerability Detail

When calculating the LP price for weight pools, the invariant of the pool is required. The current implementation looks up the *last* variant instead of the *current* invariant, which is different because last variant is only updated upon join/exits and not swaps.

https://github.com/sherlock-audit/2024-11-majora/blob/main/portal/contracts/facets/MajoraPortalBalancerOracleFacet.sol#L76-L97

Impact

LP price is not accurately calculated, which would impact the result oracleEnter/oracleExit of balancer deposit block.

Recommendation

Use current invariant instead of last one. It can be queried by `pool.getInvariant()`: https://github.com/balancer/balancer-v2-monorepo/blob/master/pkg/pool-weighted/contracts/BaseWeightedPool.sol#L88.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/portal/pull/1/commits/b9fd804a621e0fc0de0edc142eae8764c001a5d9

Issue L-4: It is recommended to remove unnecessary storage reads.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/179

Summary

It is recommended to remove unnecessary storage reads.

Vulnerability Detail

https://github.com/sherlock-audit/2024-11-majora/blob/c057daa73301a00d13b5e67141e8cf996db152f5/portal/contracts/libraries/UsingDiamondAuthority.sol#L14-L14

```
    modifier restricted() {
@>        LibAccess.AccessStorage storage ds = LibAccess.accessStorage();
        _checkCanCall(LibRelayer.msgSender(), LibRelayer.msgData());
        _;
    }
```

The storage ds is not used here; it is recommended to remove it.

Impact

Waste of gas.

Recommendation

remove unnecessary storage reads

Discussion

ZeroTrust01

fixed: cb6673b

Issue L-5: It is recommended to remove unused base classes.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/178

Summary

It is recommended to remove unused base classes.

Vulnerability Detail

https://github.com/sherlock-audit/2024-11-majora/blob/c057daa73301a00d13b5e67141e8cf996db152f5/portal/contracts/facets/MajoraPortalSwapRouterFacet.sol#L23-L23

https://github.com/sherlock-audit/2024-11-majora/blob/c057daa73301a00d13b5e67141e8cf996db152f5/portal/contracts/facets/MajoraPortalXReceiverFacet.sol#L20-L20

```
contract MajoraPortalSwapRouterFacet is UsingDiamondOwner {
```

It's better to remove this base class UsingDiamondOwner, especially if you're planning to replace onlyOwner with restricted. The same situation applies to MajoraPortalXReceiverFacet.sol.

Impact

Waste of gas.

Recommendation

It's better to remove this base class UsingDiamondOwner

Discussion

ZeroTrust01

fixed: ad3e12e

Issue L-6: `MajoraAaveV3BorrowBlock#oracleEnter` fails to account for current health factor and will return an incorrect value when `hf != desired`

Source: https://github.com/sherlock-audit/2024-11-majora/issues/177

Summary

`MajoraAaveV3BorrowBlock#oracleEnter` fails to account for current health factor when estimating `debt` and will return an incorrect value when `hf != desired`

Vulnerability Detail

MajoraAaveV3BorrowBlock.sol#L234-L239

```
uint256 borrowAmountFor = IMajoraAaveV3BorrowPositionManager(
    parameters.positionManager
).borrowAmountFor(amountToDeposit);

oracleState.removeTokenAmount(parameters.collateral, amountToDeposit);
oracleState.addTokenAmount(parameters.debt, borrowAmountFor);
```

We see above that `borrowAmountFor` is used directly to determine the amount of debt that will be added to the vault. However this fails to account for the fact that `positionManager` may not (and most likely won't) be at the desired health factor. The result is that `debt` amount will be incorrect. When below desired health factor, it will return too much and when above it will return too little. Given that the `AAVEBlock` is typically early in the flow this will cause a large amount of propagating error in later calculations.

Impact

Compounding error in estimation due to largely incorrect estimation

Code Snippet

MajoraAaveV3BorrowBlock.sol#L234-L239

Tool used

Manual Review

Recommendation

Account for the current health factor during estimation

Discussion

IAm0x52

Fix confirmed: f822df1

Issue L-7: MajoraGammaUniDepositBlock# sharesToToken fails to apply portal swap fees leading to overestimate of LP value

Source: https://github.com/sherlock-audit/2024-11-majora/issues/175

Summary

When converting from token0 -> token1 and vice versa MajoraGammaUniDepositBlock#sharesToToken fails to apply portal swap fees leading to overestimate of LP value in token.

Vulnerability Detail

MajoraGammaUniDepositBlock.sol#L547-L557

```
if (parameters.token == parameters.token0) {
    tokenvalue =
        (ORACLE_FEE * portal.getOracleRate(parameters.token1, parameters.token,
        ↪  token1Amount)) /
        10000 +
        token0Amount;
} else {
    tokenvalue =
        (ORACLE_FEE * portal.getOracleRate(parameters.token0, parameters.token,
        ↪  token0Amount)) /
        10000 +
        token1Amount;
}
```

We see above that when the other token in the pair is converted to our desired token, the function fails to apply the portal swap fee. This overestimates the amount of token that will received leading to an inaccurate measurement of vault worth.

Impact

The value of the LP in token will be overestimated

Code Snippet

MajoraGammaUniDepositBlock.sol#L547-L557

Tool used

Manual Review

Recommendation

Get fee from portal and apply after the oracle conversion

Discussion

IAm0x52

Fix confirmed: 59bf7fb

Issue L-8: MajoraGammaQuickDepositBlock#oracleEnter makes redundant calls to checkPrices

Source: https://github.com/sherlock-audit/2024-11-majora/issues/174

Summary

MajoraGammaQuickDepositBlock#oracleEnter makes redundant calls to `checkPrice`

Vulnerability Detail

MajoraGammaQuickDepositBlock.sol#L419-L442

```
function oracleEnter(
    DataTypes.OracleState memory _before,
    bytes memory _parameters
) external view returns (DataTypes.OracleState memory) {

    ...

    checkPrices(params.token0, params.token1, params.pool); <- @audit first call in
    ↪  flow

    ...

    uint256 mintedShares;
    if (params.token == params.token0) {
        mintedShares = handleToken0Deposit(params, amountToDeposit);
    } else {
        mintedShares = handleToken1Deposit(params, amountToDeposit);
    }
```

MajoraGammaQuickDepositBlock.sol#L450-L520

```
function handleToken0Deposit(
    BlockParameters memory params,
    uint256 amountToDeposit
) internal view returns (uint256 mintedShares) {
    checkPrices(params.token0, params.token1, params.pool); <- @audit second call
    ↪  in flow

    ...
}
```

```
function handleToken1Deposit(
    BlockParameters memory params,
    uint256 amountToDeposit
) internal view returns (uint256 mintedShares) {
    checkPrices(params.token0, params.token1, params.pool); <- @audit second call
     ↪  in flow

    ...
}
```

We see above that checkPrices are first called at the start of oracleEnter then again during handleToken0Deposit/handleToken0Deposit later in the flow. Since nothing has changed between calls this is redundant

Impact

Excess unnecessary calls

Code Snippet

MajoraGammaQuickDepositBlock.sol#L450-L520

MajoraGammaUniDepositBlock.sol#L448-L518

Tool used

Manual Review

Recommendation

Remove checkPrices call in a least one of the current spots

Discussion

IAm0x52

Fix confirmed.

Quickswap commit: 8e4fbe Uniswap commit: e8bd944

Issue L-9: The condition for validating the index in getVestingIdAtIndex() is incorrect.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/171

Summary

The condition for validating the index in getVestingIdAtIndex() is incorrect.

Vulnerability Detail

https://github.com/sherlock-audit/2024-11-majora/blob/c057daa73301a00d13b5e67141e8cf996db152f5/fees/contracts/MajoraTokenVesting.sol#L222-L222

```
 function getVestingIdAtIndex(
        uint256 index
    ) external view returns (bytes32) {
@>        if(index < getVestingSchedulesCount()) revert IndexOutOfBounds();
        return vestingSchedulesIds[index];
    }
```

The condition for validating the index is written incorrectly, which will render the function unusable.

Impact

This causes the function to become unusable.

Recommendation

if(index >= getVestingSchedulesCount()) revert IndexOutOfBounds();

Discussion

ZeroTrust01

fixed: 8451cdb

Issue L-10: MajoraAaveV3LeveragePositionManager may end up in a worse health factor if debtToken is not enough during unleverage.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/165

Summary

MajoraAaveV3LeveragePositionManager may end up in a worse health factor if debtToken is not enough during unleverage.

Vulnerability Detail

The unleverage logic always tries to repay a *percent* amount of debt, and withdraw a *percent* amount of collateral.

However, if the amount of debtTokens is not enough, the transaction wouldn't revert, and we end up repaying less than expected. But the collateral still withdraws the original amount, and the position manager would be at a worse health factor.

https://github.com/sherlock-audit/2024-11-majora/blob/main/borrow-module/contracts/aave/MajoraAaveV3LeveragePositionManager.sol#L287

```
        uint256 debtTokenBalance = _position.debt.token.balanceOf(address(this));
        uint256 initalCollateralAmount =
 ↪  _position.collateral.aToken.balanceOf(address(this));
        uint256 initalDebtAmount =
 ↪  _position.debt.debtToken.balanceOf(address(this));
        uint256 initalCollatDebtRatio = initalCollateralAmount * 1 ether /
 ↪  initalDebtAmount;
        uint256 amountToRepay = initalDebtAmount * _percent / 10000;
        uint256 remainingDebt = initalDebtAmount - amountToRepay;
        uint256 amountToWithdraw = initalCollateralAmount - (initalCollatDebtRatio
 ↪  * remainingDebt / 1 ether);

        uint256 repayAmount;
        if(_percent < 10000) {
@>          repayAmount = debtTokenBalance > amountToRepay ? amountToRepay :
 ↪  debtTokenBalance;
        } else {
            repayAmount = debtTokenBalance;
        }
        _repay(
            address(_position.debt.token),
            repayAmount,
            _position.debt.debtType,
            address(this)
        );

        _withdraw(address(_position.collateral.token), amountToWithdraw,
↪   address(this));
```

Impact

During unleverage action, if the debtToken is not enough to repay, the position manager would be at a worse health factor state.

Recommendation

If debtToken balance is not enough for repay, revert the transaction.

Discussion

pkqs90

Fixed in https://github.com/majora-finance/borrow-module/commit/dfba57c395c9bdd7dc8a6114fec763786f3fb6b1

Issue L-11: MajoraPortalBalancerOracleFacet does not calculate accurate price for ComposableStable pools.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/164

Summary

MajoraPortalBalancerOracleFacet does not calculate accurate price for ComposableStable pools.

Vulnerability Detail

https://github.com/sherlock-audit/2024-11-majora/blob/main/portal/contracts/facets/MajoraPortalBalancerOracleFacet.sol#L154

The current price calculation for `balancerComposableLpPrice` is inaccurate, and over complicated.

```
function balancerComposableLpPrice(
    address lpTokenPair,
    address denominationToken
) public view returns (uint256 lpTokenPrice);
```

This function aims to return the price of BPT token under denominationToken.

A more simple and accurate solution is:

1. Calculate BPT/underlying by `IComposableStablePool(lpTokenPair).getRate()`
2. Calculate denominationToken/underlying by `IComposableStablePool(lpTokenPair).getTokenRate(denominationToken)`

and divide the two numbers.

Impact

MajoraPortalBalancerOracleFacet does not calculate accurate price for ComposableStable pools.

Recommendation

Provided above.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/portal/pull/1/files

Issue L-12: MajoraGammaUniDepositBlock-/MajoraGammaQuickDepositBlock `oracleEnter()` estimates token differently than `dynamicParamsInfo()`.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/163

Summary

MajoraGammaUniDepositBlock/MajoraGammaQuickDepositBlock `oracleEnter()` estimates token differently than `dynamicParamsInfo()`. This may lead to inaccurate OracleState result when simulating strategies.

Vulnerability Detail

Following shows the difference between `dynamicParamsInfo()` and `oracleEnter()` function. The main difference is when calculating tokens used for pairing up, the former uses an average amount, while the latter uses the maximum amount.

dynamicParamsInfo(): https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/uni/MajoraGammaUniDepositBlock.sol#L129-L150

```
        (uint256 token1RequiredInitialMin, uint256 token1RequiredInitialMax) =
 ↪  uniProxy.getDepositAmount(
                parameters.hypervisor,
                parameters.token,
                amountToDeposit
        );

@>      uint256 token0ToSell = (ORACLE_FEE *
                portal.getOracleRate(
                    parameters.token1,
                    parameters.token0,
                    (token1RequiredInitialMin + token1RequiredInitialMax) / 2
                )) / 10000;

        (uint256 optimalToken0ToSell, ) = findOptimalTokenToSell(amountToDeposit,
 ↪  token0ToSell);

        // Get the new balance of token1 after finding the optimal deposit amount
        (uint256 token1requiredMin, uint256 token1RequiredMax) =
 ↪  uniProxy.getDepositAmount(
                address(parameters.hypervisor),
            parameters.token,
            optimalToken0ToSell
        );

        swap = DataTypes.DynamicSwapParams({
            fromToken: parameters.token,
            toToken: parameters.token1,
@>          value: (token1requiredMin + token1RequiredMax) / 2,
            valueType: DataTypes.SwapValueType.OUTPUT_STRICT_VALUE
        });
```

oracleEnter(): https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/uni/MajoraGammaUniDepositBlock.sol#L448-L482

```
    function handleToken0Deposit(
        BlockParameters memory params,
        uint256 amountToDeposit
    ) internal view returns (uint256 mintedShares) {
        checkPrices(params.token0, params.token1, params.pool);

        (, uint256 token1BalAfterDeposit) =
↪   uniProxy.getDepositAmount(params.hypervisor, params.token, amountToDeposit);

        // Check if the token1 balance is zero
        if (token1BalAfterDeposit == 0) {
            return 0;
        }

        // Calculate the amount of token0 to sell using the oracle rate
@>      uint256 token0ToSell = (ORACLE_FEE *
            portal.getOracleRate(params.token1, params.token0,
↪   token1BalAfterDeposit)) / 10000;

        // Find the optimal amount of token0 to sell and the corresponding amount
↪   of token1
        (uint256 optimalToken0ToSell, ) = findOptimalTokenToSell(amountToDeposit,
↪   token0ToSell);

        // Get the new balance of token1 after finding the optimal deposit amount
        (, uint256 newToken1BalAfterDeposit) = uniProxy.getDepositAmount(
            address(params.hypervisor),
            params.token,
            optimalToken0ToSell
        );

        // Update the amount of token0 to sell based on the new balance
        token0ToSell =
@>          (ORACLE_FEE * portal.getOracleRate(params.token1, params.token0,
↪   newToken1BalAfterDeposit)) /
```

```
            10000;

        // Estimate minted shares and update oracle state
        mintedShares = _estimateMintedShares(optimalToken0ToSell,
    ↪   newToken1BalAfterDeposit, params);
    }
```

Impact

Inaccurate OracleState result when simulating strategies.

Recommendation

Change `oracleEnter()` implementation to be the same as `dynamicParamsInfo()`.

Discussion

pkqs90

Added comment in https://github.com/majora-finance/gamma-blocks/pull/1/files allowbreak #

pkqs90

Fix confirmed https://github.com/majora-finance/gamma-blocks/pull/1/commits/e32bd6105888f1d3eb3140b27a35f50c4df1d8dc

Issue L-13: MajoraGammaUniDepositBlock-/MajoraGammaQuickDepositBlock may DoS during LP deposit for an edge case.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/162

Summary

MajoraGammaUniDepositBlock/MajoraGammaQuickDepositBlock may DoS during LP deposit for an edge case.

Vulnerability Detail

`uniProxy.getDepositAmount` returns the minimum/maximum amount of paired tokens that should be used for deposit.

In an edge case, if we have `amountStartT == amountEndT`, and `amountToDeposit > amountStartT` the following code would calculate `token0In = amountEndT - 1`, which is out of range.

https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/uni/MajoraGammaUniDepositBlock.sol#L271

```
    (uint256 amountStartT, uint256 amountEndT) = uniProxy.getDepositAmount(
        address(parameters.hypervisor),
        parameters.token1,
        amountTokenTargetToDeposit
    );

    if (amountToDeposit > amountEndT) {
@>      token0In = amountEndT - 1;
    } else if (amountToDeposit >= amountStartT) {
        token0In = amountToDeposit;
    } else {
        revert IncorrectRatio();
    }
```

Impact

Potential DoS at a very edge case.

Recommendation

Change `token0In = amountEndT - 1;` to `token0In = amountEndT;`.

Discussion

pkqs90

Added comment in https://github.com/majora-finance/gamma-blocks/pull/1/

pkqs90

Fix confirmed: https://github.com/majora-finance/gamma-blocks/pull/1

Issue L-14: MajoraGammaUniDepositBlock-/MajoraGammaQuickDepositBlock does not estimate shares correctly in oracleEnter.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/161

Summary

In MajoraGammaUniDepositBlock/MajoraGammaQuickDepositBlock's oracleEnter, the amount of LP tokens that would be received is calculated and added in OracleState. However, this calculation is not accurate.

Vulnerability Detail

Let's see how Gamma calculates this: https://github.com/GammaStrategies/hypervisor/blob/master/contracts/Hypervisor.sol#L125

```
    function deposit(
        uint256 deposit0,
        uint256 deposit1,
        address to,
        address from,
        uint256[4] memory inMin
    ) nonReentrant external returns (uint256 shares) {
        require(deposit0 > 0 || deposit1 > 0);
        require(deposit0 <= deposit0Max && deposit1 <= deposit1Max);
        require(to != address(0) && to != address(this), "to");
        require(msg.sender == whitelistedAddress, "WHE");

        /// update fees
        zeroBurn();

@>      (uint160 sqrtPrice, , , , , , ) = pool.slot0();
@>      uint256 price = FullMath.mulDiv(uint256(sqrtPrice).mul(uint256(sqrtPrice)),
↪   PRECISION, 2**(96 * 2));

        (uint256 pool0, uint256 pool1) = getTotalAmounts();

        shares = deposit1.add(deposit0.mul(price).div(PRECISION));

        ...
    }
```

Then how Majora calculates this:
https://github.com/sherlock-audit/2024-11-majora/blob/main/gamma-blocks/contracts/quick/MajoraGammaQuickDepositBlock.sol#L346-L348

```
    function _estimateMintedShares(
        uint256 deposit0,
        uint256 deposit1,
        BlockParameters memory parameters
    ) public view returns (uint256 estimatedShares) {
        require(deposit0 > 0 || deposit1 > 0, "Invalid deposit amounts");
        checkPrices(parameters.token0, parameters.token1, parameters.pool);

        (uint256 pendingFees0, uint256 pendingFees1) =
  ↪ AlgebraFeeCalculator.fetchAndComputePendingFees(
            parameters.hypervisor
        );

        uint256 price;
        {
            // Calculate the price based on the current tick
            (, int24 currentTick, , , , , ) =
  ↪ IAlgebraPool(parameters.pool).globalState();

@>          uint160 sqrtPrice = TickMath.getSqrtRatioAtTick(currentTick);

            // Convert sqrtPrice to uint256 and square it
            uint256 squaredPrice = uint256(sqrtPrice) * uint256(sqrtPrice);

            // Use FullMath's mulDiv function to perform the calculation
            price = FullMath.mulDiv(squaredPrice, PRECISION, 2 ** (96 * 2));
        }
        ...
    }
```

The difference is that Gamma uses `slot0.sqrtPrice` but Majora uses the `slot0.tick` and calculates the sqrtPrice according to it. This may not be a big difference because each tick is 0.01% distanced from each other, but it is nice to make the calculation the same.

Impact

Blocks will calculate an inaccurate amount of LP tokens added to OracleState during oracleEnter() function.

Recommendation

Use the slot0.sqrtPrice instead of slot0.tick.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/gamma-blocks/pull/1/files

Issue L-15: In MajoraAaveV3BorrowBlock:: dynamicParamsInfo, the calculated debt tokens are larger than the actual amount required to be repaid.This can result in dust debt tokens remaining in the BorrowPositionManager contract indefinitely.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/155

Summary

In MajoraAaveV3BorrowBlock::dynamicParamsInfo, the calculated debt tokens are larger than the actual amount required to be repaid.This can result in dust debt tokens remaining in the BorrowPositionManager contract indefinitely.

Vulnerability Detail

```
function dynamicParamsInfo(
        DataTypes.BlockExecutionType _exec,
        bytes memory _params,
        DataTypes.OracleState memory _oracleData,
        uint256 _percent
    ) external view returns (bool, DataTypes.DynamicParamsType, bytes memory) {
       //.........
        if (_percent == 10000) {
            if (status.positionDeltaIsPositive) {//debt token -> collateral token
                swap = DataTypes.DynamicSwapParams({
                    fromToken: parameters.debt,
                    toToken: parameters.collateral,
                    valueType: DataTypes.SwapValueType.INPUT_STRICT_VALUE,
                    value: status.deltaAmount * 9999 / 10000
                });
            } else {//collateral token -> debt token
                //avoid issue when the debt is a little bit higher than the token
↪   amount (< 0.01%)
@>>                 uint256 toBuy = status.deltaAmount * 10001 / 10000;
                if(toBuy < status.debtAmount / 10_000) {
@>>                     toBuy *= 25;
                }
                swap = DataTypes.DynamicSwapParams({
                    fromToken: parameters.collateral,
                    toToken: parameters.debt,
                    valueType: DataTypes.SwapValueType.OUTPUT_STRICT_VALUE,
                    value: toBuy
                });
            }

            return (
                true,
                DataTypes.DynamicParamsType.PORTAL_SWAP,
                abi.encode(swap)
            );
        }
    }
```

The root cause is that toBuy is always greater than the actual debt. Moreover, the dust tokens are not withdrawed during the final exit block. Although these are dust tokens, for high-value tokens like BTC, even dust tokens can represent a significant amount of funds.

Impact

Dust debt tokens may remain in the BorrowPositionManager contract forever, resulting in minor financial losses.

Recommendation

Add a function to allow the extraction of dust tokens.

Discussion

ZeroTrust01

fixed: 088a560

Issue L-16: Attackers can increase victim's StkMAJ cooldown period by sending them tokens

Source: https://github.com/sherlock-audit/2024-11-majora/issues/148

Summary

In StkMAJ token, the cooldown period is updated for the recipient upon each token transfer. An attack can perform a grief attack by transfering tokens to a victim to increase their cooldown period.

Vulnerability Detail

Example scenario: User A has X StkMAJ with 1 day of cooldown period, User B has a bunch of StkMAJ tokens with 30 day cooldown. User B can send X/30 StkMAJ tokens to increase by ~1 day.

https://github.com/sherlock-audit/2024-11-majora/blob/main/fees/contracts/stkMAJ/StkMAJ.sol#L156

```
    // The following functions are overrides required by Solidity.
    function _update(address from, address to, uint256 amount)
        internal
        override
    {
        if (from != to) {
            uint256 balanceOfFrom = balanceOf(from);
            uint256 balanceOfTo = balanceOf(to);
            uint256 previousSenderCooldown = stakersCooldowns[from];

@>          stakersCooldowns[to] = getNextCooldownTimestamp(previousSenderCooldown,
 ↪  amount, to, balanceOfTo);
            // if cooldown was set and whole balance of sender was transferred -
 ↪  clear cooldown
            if (balanceOfFrom == amount && previousSenderCooldown != 0) {
                stakersCooldowns[from] = 0;
            }
        }

        super._update(from, to, amount);
    }
```

However, considering the attacker would have to use non trivial amount of tokens to perform a grief attack, this is reported as a low severity issue.

Impact

Attackers can increase other user's cooldown period by sending them tokens.

Recommendation

It is hard to mitigate this issue without changing the current design, e.g. change token transfer to a two-step transfer.

Considering this is the same mechanism that stkAave token uses, it may okay for such issue to exist as long as it's publicly disclosed to users.

Discussion

pkqs90

Sponsor acknowledged. Won't fix.

Issue L-17: MajoraPortalSwapHarvestBlock `oracleHarvest()` function should use dynamic swapFeesBps instead of a constant value.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/147

Summary

When calculating the amount of tokens that will be received for swap in MajoraPortalSwapHarvestBlock's `oracleHarvest()` function, a 99.5% contant is used. However, the swap fee is actually a dynamic fee `IMajoraPortal(portal).swapFeesBps()` that can be changed in the portal.

Vulnerability Detail

- https://github.com/sherlock-audit/2024-11-majora/blob/main/majora-blocks/contracts/MajoraPortalSwapHarvestBlock.sol#L9

```
    function oracleHarvest(
        DataTypes.OracleState memory _before,
        bytes memory _parameters
    ) external view returns (DataTypes.OracleState memory) {
        BlockParameters memory parameters = abi.decode(_parameters,
 ↪  (BlockParameters));
        DataTypes.OracleState memory oracleState = _before;

        uint256 amountToSwap = (oracleState.findTokenAmount(parameters.token) *
 ↪  parameters.tokenInPercent) / 10000;
        if (amountToSwap == 0 && !oracleState.tokenExists(parameters.tokenOut)) {
            amountToSwap = IERC20(parameters.tokenOut).balanceOf(oracleState.vault);
            if (amountToSwap > 0) oracleState.addTokenAmount(parameters.tokenOut,
 ↪  amountToSwap);
        }

@>      uint256 received = (portal.getOracleRate(parameters.token,
 ↪  parameters.tokenOut, amountToSwap) * 995) / 1000;

        oracleState.removeTokenAmount(parameters.token, amountToSwap);
        oracleState.addTokenAmount(parameters.tokenOut, received);
        return oracleState;
    }
```

Impact

Amount of received tokens may be inaccurate.

Recommendation

Use `IMajoraPortal(portal).swapFeesBps()` to calculate swap fees instead.

Discussion

IAm0x52

Fix: cf7c8fb

Issue L-18: MajoraStargateDeposit/MajoraStargateDepositAndStakeLPBlock does not perform dedusting logic for `oracleEnter()`/`oracleExit()` functions.

Source: https://github.com/sherlock-audit/2024-11-majora/issues/142

Summary

MajoraStargateDeposit/MajoraStargateDepositAndStakeLPBlock does not perform dedusting logic for `oracleEnter()`/`oracleExit()` functions.

Vulnerability Detail

StargatePool deposits and redeems both have a de-dusting logic. This is commonly seen in bridges to generalize decimals. For example, the StargateMetis Pool, Metis has 18 decimals but sharedDecimal in StargatePool is only 6. So when users deposit (1e18+1e10) Metis, they will only get 1e18 LP tokens. The same for redeem.

https://github.com/stargate-protocol/stargate-v2/blob/main/packages/stg-evm-v2/src/StargatePool.sol#L55-L92

```
    function deposit(
        address _receiver,
        uint256 _amountLD
    ) external payable nonReentrantAndNotPaused returns (uint256 amountLD) {
        // charge the sender
        _assertMsgValue(_amountLD);
@>      uint64 amountSD = _inflow(msg.sender, _amountLD);
        _postInflow(amountSD); // increase the local credit and pool balance

        // mint LP token to the receiver
@>      amountLD = _sd2ld(amountSD);
        lp.mint(_receiver, amountLD);
        tvlSD += amountSD;
        emit Deposited(msg.sender, _receiver, amountLD);
    }
    function redeem(uint256 _amountLD, address _receiver) external
 ↪  nonReentrantAndNotPaused returns (uint256 amountLD) {
@>      uint64 amountSD = _ld2sd(_amountLD);
        paths[localEid].decreaseCredit(amountSD);

        // de-dust LP token
@>        amountLD = _sd2ld(amountSD);
        // burn LP token. Will revert if the sender doesn't have enough LP token
        lp.burnFrom(msg.sender, amountLD);
        tvlSD -= amountSD;

        // send the underlying token from the pool to the receiver
        _safeOutflow(_receiver, amountLD);
        _postOutflow(amountSD); // decrease the pool balance

        emit Redeemed(msg.sender, _receiver, amountLD);
    }
```

The issue here is that for `oracleEnter()/oracleExit()` functions, the same dedusting logic should be performed when adding tokens to OracleState.

- https://github.com/sherlock-audit/2024-11-majora/blob/main/stargate-blocks/contracts/MajoraStargateDepositLPBlock.sol#L76
- https://github.com/sherlock-audit/2024-11-majora/blob/main/stargate-blocks/contracts/MajoraStargateDepositAndStakeLPBlock.sol#L91

Impact

OracleState is not correctly calculated.

Recommendation

Implement a dedusting logic for `oracleEnter()/oracleExit()`.

Discussion

pkqs90

Fix confirmed: https://github.com/majora-finance/stargate-blocks/pull/1/commits/a8a53b47ec0e0dac0c3a37c9896afecb4a05472e

Disclaimers