TapiocaDAO Security review

Pashov Audit Group

Conducted by: Peakbolt, carrotsmuggler, tsvetanovv, Stoicov, 0xWeiss, Nisedo

November 27th 2023 - December 22nd 2023

Contents

1. About Pashov Audit Group

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

2. Disclaimer

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

3. Introduction

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

4. About tapioca-periph

The `tapioca-periph` repository contains external contracts that are not used as core functionalities of the protocol but are still needed in order for it to work. It includes transfer helper contracts, transaction bundling mechanisms, whitelisting mechanism, swapping and oracles logic.

5. Risk Classification

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

5.1. Impact

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

5.2. Likelihood

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

5.3. Action required for severity levels

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

6. Security Assessment Summary

***review commit hash* - 57ff863ae205a968d3ede03d9e04606b60603ee7**

***fixes review commit hash* -**

Scope

The following smart contracts were in scope of the audit:

- `StargateLbpHelper`
- `Magnetar/modules/MagnetarMarketModule`
- `Magnetar/MagnetarHelper`
- `Magnetar/MagnetarV2`
- `Magnetar/MagnetarV2Storage`
- `Swapper/UniswapV2Swapper`
- `Swapper/UniswapV3Swapper`
- `Swapper/BaseSwapper`
- `oracle/Seer`
- `oracle/utils/SequencerCheck`
- `oracle/implementations/Arbitrum/GLPOracle`
- `oracle/implementations/Arbitrum/TapOracle`
- `oracle/Cluster/Cluster`

7. Executive Summary

Over the course of the security review, Peakbolt, carrotsmuggler, tsvetanovv, Stoicov, 0xWeiss, Nisedo engaged with TapiocaDAO to review tapioca-periph. In this period of time a total of **36** issues were uncovered.

Protocol Summary

Protocol Name	tapioca-periph
Repository	https://github.com/Tapioca-DAO/tapioca-periph
Date	November 27th 2023 - December 22nd 2023
Protocol Type	periphery contracts

Findings Count

Severity	**Amount**
Critical	3
High	7
Medium	16
Low	10
Total Findings	**36**

Summary of Findings

ID	Title	Severity	Status
[C-01]	The read() function returns the wrong value because it is not properly scalable	Critical	Resolved
[C-02]	Anyone can update the cluster address	Critical	Resolved
[C-03]	Approvals can be exploited due to insufficient checks in _exitPositionAndRemoveCollateral	Critical	Resolved
[H-01]	StargateLbpHelper.participate() will send tokens to the wrong address	High	Resolved
[H-02]	StargateLbpHelper.sgReceive() could encounter permanent error that causes received tokens to be stuck in contract	High	Resolved
[H-03]	Multisig wallets that call StargateLbpHelper.participate() could cause received token to be stolen	High	Resolved
[H-04]	Use of only higher price in Seer makes it vulnerable to price manipulation	High	Resolved
[H-05]	Incorrect address for TOFT debit	High	Resolved
[H-06]	Incorrect leveraging operations call in Magnetar	High	Resolved
[H-07]	Underpaying/Overpaying of Stargate fee will occur in StargateLbpHelper.participate()	High	Resolved
[M-01]	TickMath and FullMath libraries are missing unchecked	Medium	Resolved
[M-02]	Magnetar exitPositionAndRemoveCollateral() could fail when withdrawing removed assets/collateral	Medium	Resolved
[M-03]	Lack of sequencer uptime check for TapOracle can cause stale prices	Medium	Resolved
[M-04]	UniswapV3Swapper should not use the same poolFee for all token pairs	Medium	Resolved
[M-05]	UniswapV3Swapper and UniswapV2Swapper will fail when depositing received ETH into	Medium	Resolved

	YieldBox		
[M-06]	Using buildSwapData() with tokenIn/tokenOut will cause swap to fail when depositing into/withdrawing from YieldBox	Medium	Resolved
[M-07]	FETCH_TIME is too large and this can lead to a stale price	Medium	Resolved
[M-08]	Missing implementation of the quoteLayerZeroFee() in StargateLbpHelper.sol	Medium	Resolved
[M-09]	_withdrawToChain() will refund excess gas for LZ call to the wrong address	Medium	Resolved
[M-10]	TapOracle TWAP duration for Uniswap Oracle should be at least 30 mins	Medium	Resolved
[M-11]	Missing gas forwarding in cross-chain call	Medium	Resolved
[M-12]	Not maximizing Glp in GlpOracle miscalculates _minGlp when adding liquidity	Medium	Resolved
[M-13]	Any tokens or eth in Magnetar can be drained	Medium	Resolved
[M-14]	Magnetar _depositRepayAndRemoveCollateralFromMarket can fail due to rounding errors	Medium	Resolved
[M-15]	Using only DEX swappers is suboptimal	Medium	Resolved
[M-16]	Incorrect order when calculating the transferred amount to Stargate makes the participate() function unusable	Medium	Resolved
[L-01]	Redeem functions and retryRevert() in StargateLbpHelper are unnecessary	Low	Resolved
[L-02]	getOutputAmount() should not be used in state-changing functions	Low	Acknowledged
[L-03]	Incorrect interfaceId check in _withdrawToChain()	Low	Resolved
[L-04]	Protocol is using old version of OpenZeppelin contracts	Low	Resolved
[L-05]	answeredInRound is deprecated	Low	Resolved
[L-06]	zroPaymentAddress should not be hardcoded to	Low	Resolved

	address(0)		
[L-07]	StargateLbpHelper.participate() could leave dust amount in contract	Low	Resolved
[L-08]	Un-used ParticipateData.poolId for StargateLbpHelper	Low	Resolved
[L-09]	Useless payload check	Low	Resolved
[L-10]	Do not hardcode _payInZRO to false when estimating layerZero fees	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] The `read()` function returns the wrong value because it is not properly scalable

Severity

Impact: High, because price data will be very inaccurate.

Likelihood: High, because it will always happen.

Description

In `OracleUniSolo.sol` we have the function `read()`:

```
function read() external view override returns (uint256) {
        return _readUniswapQuote(10 ** inBase);
    }
```

This function returns the current exchange rate between two tokens from Uniswap. It's also extremely important because provides price data. But there is a big mistake in the argument.

You can see how is `inBase` variable:

```
inBase = 10 ** (inCur.decimals());
```

The correct version of the `read()` function should be using `inBase` directly as the argument to `_readUniswapQuote()`. The `inBase` is already calculated as `10 ** (inCur.decimals())`.

Because of the wrong value passed to `_readUniswapQuote()` the consequences can be huge. In one case you may get an overflow and the function may not work at all, in the other case it will return a totally wrong value.

Recommendation

Pass the `inBase` directly to the function without further scaling:

```
function read() external view override returns (uint256) {
        return _readUniswapQuote(inBase);
    }
```

[C-02] Anyone can update the cluster address

Impact: High, because a malicious user would gain access to a very important function.

Likelihood: High, the lack of access control makes this function very easy to exploit.

Description

In `MagnetarV2.sol` we have `setCluster()` function:

```
function setCluster(ICluster _cluster) external {
        if (address(_cluster) == address(0)) revert NotValid();
        emit ClusterSet(cluster, _cluster);
        cluster = _cluster;
    }
```

This function updates the cluster address, which is extremely important. But this function has no access control. Only the owner should be able to call this function. But currently absolutely anyone can call `setCluster()` and change the cluster address, which can cause major harm to the protocol.

Recommendations

Add the `onlyOwner` modifier to the `setCluster()` function:

```
function setCluster(ICluster _cluster) external onlyOwner {
        if (address(_cluster) == address(0)) revert NotValid();
        emit ClusterSet(cluster, _cluster);
        cluster = _cluster;
    }
```

[C-03] Approvals can be exploited due to insufficient checks in `_exitPositionAndRemoveCollateral`

Severity

Impact: High, loss of provided liquidity tokens due to approval exploit

Likelihood: High, can be called by anyone

Description

The function `_exitPositionAndRemoveCollateral` in the MagnetarMarketModule carries out a number of operations. It unlocks locked liquidity positions, and then removes assets from the markets. The first two operations are of interest here, called `exit` and `unlock`. The caller provides boolean values to the function call which determines if these operations

are run. These boolean values are stored in `removeAndRepayData.exitData.exit` and `removeAndRepayData.unlockData.unlock`.

The issue is that for a certain combination of values passed, the user can pull liquidity tokens from other users who have given approval to the Magnetar contract. Since the Magnetar contract is a helper contract for user interactions, it is assumed that users have already given allowance of their tokens and ERC721 positions to Magnetar.

Lets look at a scenario where the attacker sets `removeAndRepayData.exitData.exit` to false, and `removeAndRepayData.unlockData.unlock` to true.

Due to the first bool being false, the function does not do the `exit` operation.

```
uint256 tOLPId = 0;
        if (removeAndRepayData.exitData.exit) { ... }
```

In the `unlock` operation, the tokenId which has to be unlocked is to be set. As the `exit` operation is skipped, the values stored in `tOLPId` is still 0 as shown above.

```
if (removeAndRepayData.unlockData.tokenId != 0) {
                if (tOLPId != 0) {
                    if (tOLPId != removeAndRepayData.unlockData.tokenId)
                        revert tOLPTokenMismatch();
                }
                tOLPId = removeAndRepayData.unlockData.tokenId;
            }
```

Here, the attacker sets the victim's token id in the field `removeAndRepayData.unlockData.tokenId`. Since its non-zero, the inner if clause is reached. Since `tOLPId` is still 0 due to skipping the `exit` operation, the if clause is skipped, and the `tOLPId` is set to `removeAndRepayData.unlockData.tokenId`.

The issue is that there is no check whether the caller actually owns this tokenId! in the next line, `unlock` is called on this tokenId, and the proceeds are sent to `user`, who is the attacker.

```
ITapiocaOptionLiquidityProvision(
                removeAndRepayData.unlockData.target
            ).unlock(tOLPId, externalData.singularity, user);
```

If the victim has given approval of their liquidity token to Magnetar in the past, which is likely if they have ever interacted with Magnetar, they would lose their token and the liquidity would be unlocked and sent off to the attacker. So the attacker can hijack anyone's liquidity token. The magnetar contract does not even need to own this ERC721 token, because as we can see from the `unlock` function, the token is burnt directly from the holder's wallet as long as Magnetar has aproval.

```
require(
            _isApprovedOrOwner(msg.sender, _tokenId),
            "tOLP: not owner nor approved"
        );

        _burn(_tokenId);
```

Recommendations

Add a user check similar to the one present in `exit` operation.

```
address ownerOfTapTokenId = IERC721(oTapAddress).ownerOf(
                removeAndRepayData.unlockData.tokenId
            );

            if (ownerOfTapTokenId != user && ownerOfTapTokenId != address(this))
                revert NotValid();
```

8.2. High Findings

[H-01] `StargateLbpHelper.participate()` will send tokens to the wrong address

Severity

Impact: Med, as the tokens incorrectly sent to `msg.sender` may not be recoverable if the caller do not have control over it (when caller is using a multisig wallet contract)

Likelihood: High, as the `StargateLbpHelper` will perform a stargate swap to the wrong address

Description

`StargateLbpHelper.participate()` is called on the non-host chain to send tokens via Stargate to `StargateLbpHelper` on host chain for depositing into Balancer Liquidity Bootstrapping Pool.

This will cause an issue for `participate()` as it will perform Stargate `router.swap()` to destination address `msg.sender` instead of the `StargateLbpHelper` contract on host chain.

```
function participate(
        StargateData calldata stargateData,
        ParticipateData calldata lbpData
    ) external payable nonReentrant {
        IERC20 erc20 = IERC20(stargateData.srcToken);

        // retrieve source token from sender
        erc20.safeTransferFrom(msg.sender, address(this), stargateData.amount);

        // compute min amount to be received on destination
        uint256 amountWithSlippage = stargateData.amount -
            ((stargateData.amount * stargateData.slippage) /
                SLIPPAGE_PRECISION);

        // approve token for Stargate router
        erc20.safeApprove(address(router), stargateData.amount);

        // send over to another layer using the Stargate router
        router.swap{value: msg.value}(
            stargateData.dstChainId,
            stargateData.srcPoolId,
            stargateData.dstPoolId,
            payable(msg.sender), //refund address
            stargateData.amount,
            amountWithSlippage,
            IStargateRouterBase.lzTxObj({
                dstGasForCall: 0,
                dstNativeAmount: 0,
                dstNativeAddr: "0x0"
            }),
            abi.encodePacked
            //(msg.sender), //@audit this should be a parameter to `StargateLbpHelper` address on h
            abi.encode(lbpData, msg.sender)
        );
    }
```

Recommendations

Allow `participate()` to pass in the receiving address of `StargateLbpHelper` on the host chain.

Note that `StargateLbpHelper` will be deployed using `CREATE2` via `TapiocaDeployer`, the contracts will not have the same address on the host chain and non-host chains. That is because the constructor parameters for `StargateLbpHelper` will be different for them, causing the deployed address to be different as CREATE2 precompute address based on creationcode that includes constructor parameters. So it is important to use a parameter for the destination `StargateLbpHelper` address and not hardcode it.

[H-02] `StargateLbpHelper.sgReceive()` could encounter permanent error that causes received tokens to be stuck in contract

Severity

Impact: High, as tokens will be stuck in contract

Likelihood: Medium, as it only occur for permanent errors

Description

`StargateLbpHelper.sgReceive()` receives tokens that are sent across chains via Stargate swap, and then swaps them via LBP pool. `StargateLbpHelper` has a `retryRevert()` to allow the owner to retry the execution on stargate swap failure.

However, if the failure is due to a permanent error, `retryRevert()` will not help with the recovery. When that happens, the tokens received will be stuck in the `StargateLbpHelper` contract, with no means to retrieve them.

Recommendations

Either allow token recipient to retrieve the tokens or transfer to recipient when such a permanent error occurs.

[H-03] Multisig wallets that call `StargateLbpHelper.participate()` could cause received token to be stolen

Severity

Impact: High, as received tokens on destination chain will be lost

Likelihood: Medium, this occurs when caller is a multisig wallet contract

Description

`StargateLbpHelper.participate()` is used to perform a cross chain transfer to the host chain and then a swap using the Balancer Liquidity Bootstrapping Pool.

The issue is that the receiver address of the LBP swap on the destination chain is hardcoded to `msg.sender` (passed in by `participate()`).

This will be a problem if `msg.sender` is a multisig wallet contract, as the wallet owners may not have control over the same address as `msg.sender` on the destination chain, causing the received tokens to be lost (if the recipient is not willing to return the funds).

In the worst case, the tokens could be sent to an undeployed address and an attacker could seize the opportunity to possibly steal the received tokens by taking control of the `msg.sender` address on destination chain (see Wintermute hack article).

```
function participate(
        StargateData calldata stargateData,
        ParticipateData calldata lbpData
    ) external payable nonReentrant {
       ...
        router.swap{value: msg.value}(
            stargateData.dstChainId,
            stargateData.srcPoolId,
            stargateData.dstPoolId,
            payable(msg.sender),
            stargateData.amount,
            amountWithSlippage,
            IStargateRouterBase.lzTxObj({
                dstGasForCall: 0,
                dstNativeAmount: 0,
                dstNativeAddr: "0x0"
            }),
            abi.encodePacked(msg.sender),
            abi.encode
            //(lbpData, msg.sender) //@audit receiver address on destination chain should be a para
        );


    function sgReceive(
        uint16,
        bytes memory,
        uint256,
        address token,
        uint256 amountLD,
        bytes memory payload
    ) external {
        ...
        // decode payload
        (ParticipateData memory data, address receiver) = abi.decode
        //( //@audit receiver is set to msg.sender
            payload,
            (ParticipateData, address)
        );
       ...
        IBalancerVault.FundManagement memory fundManagement = IBalancerVault
            .FundManagement({
                sender: address(this),
                recipient: payable
                //(receiver), //@audit token is sent to receiver (which is msg.sender on src chain)
                fromInternalBalance: false,
                toInternalBalance: false
            });

        IERC20(data.assetIn).approve(address(lbpVault), 0);
        IERC20(data.assetIn).approve(address(lbpVault), amountLD);
        lbpVault.swap(
            singleSwap,
            fundManagement,
            data.minAmountOut,
            (data.deadline != 0 ? data.deadline : block.timestamp)
        );
    }
```

Recommendations

Allow caller to pass in the `receiver` address for the LBP swap.

[H-04] Use of only higher price in `Seer` makes it vulnerable to price manipulation

Severity

Impact: High, price manipulation will cause loss of funds

Likelihood: Medium, Uniswap pool can be price-manipulated when TWAP duration is too low or liquidity is not spread over a wide range

Description

`Seer.sol` uses the Angle Protocol's `OracleMulti.sol`, which is a combination of Chainlink and Uniswap V3 TWAP oracles. The design of `OracleMulti.sol` is to return both Chainlink and Uniswap prices, and let the protocol choose the price that is most advantageous. For example, Angle uses the lower price between Chainlink and Uniswap, for a mint transaction using collateral. (see Angle's Oracle Design)

The issue is that `Seer.sol` only uses the higher price for its price retrieval functions like `get()`, `peek()`, `peekSpot()`. It ignores the lower price and do not return it at all. That means it could use the price that is disadvantageous to the protocol.

```
function get(
        bytes calldata
    ) external virtual nonReentrant returns (bool success, uint256 rate) {
        // Checking whether the sequencer is up
        _sequencerBeatCheck();

        //@audit only the higher price is used, while ignoring the lower
        //(first returned variable)
        (, uint256 high) = _readAll(inBase);
        return (true, high);
    }
```

This causes contracts that use `Seer.sol` to be vulnerable to price manipulation attacks.

For example, when retrieving the price of payment token for exercising oTAP options, it is more advantageous to use the lower price. By using a higher price, it allows an attacker to artificially inflate the price of the payment token by manipulating the price for the Uniswap V3 pool. That means the attacker will be able to exercise the option with significantly lower amount of payment tokens.

Recommendations

Return both higher and lower values for price functions in `Seer.sol` and use the appropriate price in the protocol.

[H-05] Incorrect address for TOFT debit

Severity

Impact: High, transaction either reverts, or debits from wrong account

Likelihood: Medium, only happens when the caller is a whitelisted relayer

Description

The Magnetar contract handles the operation id `TOFT_SEND_FROM` to trigger a sending of TOFT tokens cross chain. First the parameters are extracted from the user's call, and then the actual call is done.

```
(
                    address from,
                    uint16 dstChainId,
                    bytes32 to,
                    uint256 amount,
                    ICommonOFT.LzCallParams memory lzCallParams
                ) = abi.decode(
                        _action.call[4:],
                        (
                            address,
                            uint16,
                            bytes32,
                            uint256,
                            (ICommonOFT.LzCallParams)
                        )
                    );
                _checkSender(from);

                ISendFrom(_action.target).sendFrom{value: _action.value}(
                    msg.sender,
                    dstChainId,
                    to,
                    amount,
                    lzCallParams
                );
```

As seen from the snippet, there are 2 important addresses: `msg.sender` and `from`. `from` is supposed to be the address from which the tokens are debited. `msg.sender` is the address which can have special permissions and can act as just the caller of the function. This is further supported by the implementation in the `checkSender` function.

```
function _checkSender(address _from) internal view {
        if (_from != msg.sender && !cluster.isWhitelisted(0, msg.sender))
            revert NotAuthorized();
    }
```

As we can see, either `msg.sender` and `from` are the same person, or `msg.sender` is a whitelisted address with special permissions.

However when the tokens are deducted in the `sendFrom` call, the function mistakenly deducts the token from the `msg.sender` address, instead of the `from` address. This will be an issue when msg.sender is such a whitelisted relayer, and the wrong account will have tokens debited from it. If this account has no tokens, the transaction will revert and make the whitelisted role useless.

This issue is present in multiple instances:

- [ ] action.id `TOFT_WRAP`
- [x] action.id `TOFT_SEND_FROM`
- [ ] action.id `YB_DEPOSIT_ASSET`
- [ ] action.id `MARKET_ADD_COLLATERAL`
- [ ] action.id `MARKET_BORROW`
- [ ] action.id `MARKET_LEND`
- [ ] action.id `MARKET_REPAY`
- [ ] action.id `TOFT_SEND_AND_BORROW`
- [ ] action.id `TOFT_SEND_AND_LEND`
- [ ] action.id `TOFT_DEPOSIT_TO_STRATEGY`
- [ ] action.id `TOFT_RETRIEVE_FROM_STRATEGY`

Recommendations

Deduct tokens from the `from` account instead of `msg.sender`, since `msg.sender` can just be a whitelisted relayer.

[H-06] Incorrect leveraging operations call in Magnetar

Severity

Impact: Medium, broken functionality

Likelihood: High, broken functionality never works

Description

The Magnetar contract has helper functions to interact with the Singularity and BigBang markets which also does multiple compounded operations at the same time for added functionality. One such operation which is supported by the BigBang and Singularity markets is the ability to buy and sell the collateral tokens. This is supported in the BigBang and Singularity markets as shown by the function prototypes below.

```
function buyCollateral(
        address from,
        uint256 borrowAmount,
        uint256 supplyAmount,
        bytes calldata data
    ){...}
function sellCollateral(
        address from,
        uint256 share,
        bytes calldata data
    ){...}
```

The issue is that the Magnetar contracts use the wrong interface from a previous iteration of the protocol which does not work with the current version of the contracts. This evident from the calls seen in the Magnetar contracts.

```
IMarket(_action.target).buyCollateral(
                    from,
                    borrowAmount,
                    supplyAmount,
                    minAmountOut,
                    swapper,
                    dexData
                );

IMarket(_action.target).sellCollateral(
                    from,
                    share,
                    minAmountOut,
                    swapper,
                    dexData
                );
```

As we can see from the function calls, the caller passes in 5 values but the function actually expects only 3 values in its function prototype, thus leading to broken functionality.

There are two instances of this issue in `MagnetarV2.sol`.

- [ ] action.id `MARKET_BUY_COLLATERAL`
- [ ] action.id `MARKET_SELL_COLLATERAL`

Recommendations

Change the function calls to conform to the current version of the markets.

[H-07] Underpaying/Overpaying of Stargate fee will occur in `StargateLbpHelper.participate()`

Severity

Impact: Medium, as it will underpay/overpay Stargate fee

Likelihood: High, it will always occur

Description

`StargateLbpHelper.participate()` uses Stargate `router.swap()` to send tokens to host chain for LBP swap.

However, the `_lzTxParams` parameters are hardcoded to zero, which will cause Stargate to undercharge/overcharge for the cross chain swap fee. That will occur as Stargate uses LayerZero for the cross chain messaging, which will compute the underlying LZ fee based on the `dstGasForCall`. When `dstGasForCall == 0`, it will charge the fee based on a default value of 200,000 gas for the execution in the destination chain.

That means `participate()` will overpay for the Stargate fee if it consumes < 200,000 gas for the destination chain. On the other hand, when destination chain gas consumption >

200,000 gas, it will underpay and cause the destination chain execution to fail due to OOG error.

```
function participate(
        StargateData calldata stargateData,
        ParticipateData calldata lbpData
    ) external payable nonReentrant {
        IERC20 erc20 = IERC20(stargateData.srcToken);

        // retrieve source token from sender
        erc20.safeTransferFrom(msg.sender, address(this), stargateData.amount);

        // compute min amount to be received on destination
        uint256 amountWithSlippage = stargateData.amount -
            ((stargateData.amount * stargateData.slippage) /
                SLIPPAGE_PRECISION);

        // approve token for Stargate router
        erc20.safeApprove(address(router), stargateData.amount);

        // send over to another layer using the Stargate router
        router.swap{value: msg.value}(
            stargateData.dstChainId,
            stargateData.srcPoolId,
            stargateData.dstPoolId,
            payable(msg.sender), //refund address
            stargateData.amount,
            amountWithSlippage,
            //@audit the _lzTxParams parameter should not be hardcoded to zero
            IStargateRouterBase.lzTxObj({
                dstGasForCall: 0,
                dstNativeAmount: 0,
                dstNativeAddr: "0x0"
            }),
            abi.encodePacked
            //(msg.sender), // StargateLbpHelper.sol destination address
            abi.encode(lbpData, msg.sender)
        );
    }
```

Recommendations

Pass in `_lzTxParams` parameters instead of hardcoding them.

8.3. Medium Findings

[M-01] `TickMath` and `FullMath` libraries are missing `unchecked`

Severity

Impact: Medium, as it will cause the swap to revert.

Likelihood: Medium, as it will occur when there is phantom overflow.

Description

Both `FullMath` and `TickMath` are missing `unchecked`, which causes it to incorrectly revert on phantom overflow. These libraries are supposed to handle "phantom overflow" by allowing multiplication and division even when the intermediate value overflows 256 bits as documented by UniswapV3. In the original UniswapV3 code, unchecked is not used as solidity version is < 0.8.0, which does not revert on overflow.

`TickMath` will affect `UniswapV3Swapper`, which uses `OracleLibrary` that utilizes `TickMath`. Same issue for `FullMath`, which will affect `Seer`.

Recommendations

Add in `unchecked` for both libraries.

[M-02] Magnetar `exitPositionAndRemoveCollateral()` could fail when withdrawing removed assets/collateral

Severity

Impact: Medium, as this will cause the function to revert **Likelihood:** Medium, as this occurs when user wish to withdraw the underlying collateral from YieldBox

Description

In `MagnetarMarketModule._exitPositionAndRemoveCollateral()`, the user can set `removeAndRepayData.removeCollateralFromBB = true` and `removeAndRepayData.collateralWithdrawData.withdraw = true` to remove collateral

shares from BigBang and then use those shares to withdraw the underlying collateral from YieldBox.

The amount of underlying collateral to be removed is indicated by the parameter `removeAndRepayData.collateralAmount`.

However, the actual collateral amount withdrawn could be lesser than what is provided by the parameter `removeAndRepayData.collateralAmount`. That is because a `yieldBox.toShare()` conversion is required due to `bigBang.removeCollateral()` taking in a `collateralShare` parameter.

When that occurs, it will cause `_withdraw()` to fail as the user provided `removeAndRepayData.collateralAmount` is more than what was removed from BigBang.

```
if (removeAndRepayData.removeCollateralFromBB) {
            //@audit collateralShare could be rounded down here after toShare
            //() conversion
            uint256 collateralShare = yieldBox.toShare(
                bigBang.collateralId(),
                removeAndRepayData.collateralAmount,
                false
            );
            address removeCollateralTo = removeAndRepayData
                .collateralWithdrawData
                .withdraw
                ? address(this)
                : user;

            //@audit if collateralShare is rounded down, amount removed will be
            // less than
            //       user provided removeAndRepayData.collateralAmount
            bigBang.removeCollateral(user, removeCollateralTo, collateralShare);

            //withdraw
            if (removeAndRepayData.collateralWithdrawData.withdraw) {
                bytes memory withdrawCollateralBytes = abi.encode(
                    removeAndRepayData
                        .collateralWithdrawData
                        .withdrawOnOtherChain,
                    removeAndRepayData.collateralWithdrawData.withdrawLzChainId,
                    LzLib.addressToBytes32(user),
                    removeAndRepayData
                        .collateralWithdrawData
                        .withdrawAdapterParams
                );
                _withdraw(
                    address(this),
                    withdrawCollateralBytes,
                    singularity,
                    yieldBox,
                    //@audit when amount from removeCollateral
                    //() < removeAndRepayData.collateralAmount
                       _withdraw
                    //() will fail as it tries to withdraw more than what was removed above
                    removeAndRepayData.collateralAmount,
                    true,
                    valueAmount,
                    removeAndRepayData.collateralWithdrawData.unwrap
                );
            }
        }
        _revertYieldBoxApproval(address(bigBang), yieldBox);
    }
```

The same issue also applies when `removeAndRepayData.removeAssetFromSGL = true`. Similarly for `_depositRepayAndRemoveCollateralFromMarket()` as well.

Recommendations

The amount to withdraw from YieldBox should be derived from the `collateralShare` returned by `yieldBox.toShare()`.

[M-03] Lack of sequencer uptime check for `TapOracle` can cause stale prices

Severity

Impact: Medium, oracle price will be stale **Likelihood:** Medium, occurs during period of sequencer downtime

Description

`TapOracle` takes an average of 3 TWAP prices from UniswapV3 pool with an interval of at least 4 hours (based on `FETCH_TIME`). The 3 TWAP prices are stored in `lastPrices[]` and updated when `get()` is called to retrieve TAP price.

The issue is that when the L2 sequencer is down for an extended period, there will be no interaction with the oracle via `get()`, preventing `lastPrices[]` from being updated with the latest prices. This will cause `TapOracle` to return stale prices when the sequencer recovers.

Recommendations

Add `_sequencerBeatCheck();` in the function `get()`. This is to provide a grace period when sequencer recovers from downtime for `TapOracle` to be updated with the latest prices.

It is recommended that `FETCH_TIME` be at most 1/3 of the grace period, to allow sufficient time for all 3 `lastPrices[]` to be updated when sequencer recovers.

[M-04] `UniswapV3Swapper` should not use the same `poolFee` for all token pairs

Severity

Impact: Medium, swap will fail when pool fee is wrong **Likelihood:** Medium, only occur for swap where pool fee is not 3000 (default value)

Description

UniswapV3Swapper uses the same `poolFee` for all token pairs.

The issue is that Uniswap V3 pool address is tied to both token addresses and the pool fee as shown in UniswapV3 `SwapRouter.getPool()` below. This means that the swap may fail if the token pair to swap is not available for the configured `poolFee`.

```
/// @dev Returns the pool for the given token pair and fee. The pool
    // contract may or may not exist.
    function getPool(
        address tokenA,
        address tokenB,
        uint24 fee
    ) private view returns (IUniswapV3Pool) {
        return IUniswapV3Pool(PoolAddress.computeAddress
          (factory, PoolAddress.getPoolKey(tokenA, tokenB, fee)));
    }
```

Recommendations

Allow `poolFee` to be passed in as a parameter so that the correct pool will be used for the swap.

[M-05] `UniswapV3Swapper` and `UniswapV2Swapper` will fail when depositing received ETH into YieldBox

Severity

Impact: Medium, as it causes swap to fail

Likelihood: Medium, as it occurs when depositing WETH to YieldBox

Description

When `UniswapV3Swapper.swap()` is called with `depositToYb = true` and `tokenOut = address(0)` (WETH), it will swap the Input Token for WETH and then unwrap the WETH before depositing ETH into YieldBox.

However, `yieldBox.depositAsset()` is used to deposit the unwrapped ETH, which is incorrect and will revert as it does not support deposit of native asset.

```
amountOut = _swap(
            tokenOut,
            params,
            swapData.yieldBoxData.depositToYb,
            to
        );

        if (swapData.yieldBoxData.depositToYb) {
            if (tokenOut != address(0)) {
                _safeApprove(tokenOut, address(yieldBox), amountOut);
            }
            (, shareOut) = yieldBox.depositAsset(
                swapData.tokensData.tokenOutId,
                address(this),
                to,
                amountOut,
                0
            );
        }
```

Same issue also apply for `UniswapV2Swapper`.

Recommendations

Do not unwrap WETH when `depositToYb = true`, so that WETH is directly deposited into YieldBox without unwrapping and re-wrapping. Alternatively, use `yieldBox.depositETHAsset()` instead when `tokenOut` is address(0) but this will unwrap and rewrap.

[M-06] Using `buildSwapData()` with tokenIn/tokenOut will cause swap to fail when depositing into/withdrawing from YieldBox

Severity

Impact: Medium, as `swap()` will fail

Likelihood: Medium, as it only occurs when using swap with yieldbox deposit.

Description

When `buildSwapData(address tokenIn, address tokenOut, ...)` is used to populate `SwapData`, both `tokenInId` and `tokenOutId` will be set to zero.

However, when using `swap()` with `depositToYb = true`, it will deposit to YieldBox based on the `tokenOutId`. That will fail as `tokenOutId` is zero.

Same issue for `withdrawToYb = true`, which will fail as `tokenInId` will be zero as well.

Recommendations

For `buildSwapData(address tokenIn, address tokenOut, ...)` , set `withdrawFromYb` and `depositToYb` to `false`, and remove these parameters, as it is not supported.

[M-07] `FETCH_TIME` is too large and this can lead to a stale price

Severity

Impact: High, because it might return an stale price

Likelihood: Low, it can happen in highly volatile markets

Description

The `FETCH_TIME` value in the `TapOracle.sol` contract is set to 4 hours, which determines the minimum interval between updates of the oracle price.

```
uint32 public FETCH_TIME = 4 hours;
```

But this interval is too large and can lead to an outdated price. In highly volatile markets, price data can become outdated quickly. A 4-hour interval may not be sufficient to capture important market movements, potentially leading to the oracle providing stale or inaccurate data.

Change it to 1 hour exactly as you described it in the NatSpec and in the documentation.

Recommendations

Change `FETCH_TIME` to 1 hour:

```
uint32 public FETCH_TIME = 1 hours;
```

[M-08] Missing implementation of the `quoteLayerZeroFee()` in StargateLbpHelper.sol

Severity

Impact: Low, because the user can still use `participate()` without this implementation.

Likelihood: High, because there is no implementation of this function.

Description

We use the `participate()` function when we want to do a cross-chain transfer. The function calls the `swap` method using Stargate router:

```
router.swap{value: msg.value}(
            stargateData.dstChainId,
            stargateData.srcPoolId,
            stargateData.dstPoolId,
            payable(msg.sender), //refund address
            stargateData.amount,
            amountWithSlippage,
            IStargateRouterBase.lzTxObj({
                dstGasForCall: 0,
                dstNativeAmount: 0,
                dstNativeAddr: "0x0"
            }),
            abi.encodePacked
            //(msg.sender), // StargateLbpHelper.sol destination address
            abi.encode(lbpData, msg.sender)
        );
```

To pay `fee` we send `msg.value`:

```
swap{value:msg.value}
```

From Stargate documentation we see that we have to call `quoteLayerZero()` to calculate the fee.

> For the native gas fee required for `swap()` you need to call `quoteLayerZero()` on the Router.sol contract to get the amount you should send as msg.value.

But such a function is missing. Because of this, we don't know how many fees to send for the transfer to be successful. If we send more than needed it will refund, but if we send less than needed the cross-chain transfer will fail.

Recommendations

Implement the `quoteLayerZeroFee()` function. See the documentation for more information.

[M-09] `_withdrawToChain()` will refund excess gas for LZ call to the wrong address

Severity

Impact: Medium, as refunded gas will be lost

Likelihood: Medium, happens for certain cases when performing cross chain withdrawals

Description

As `MagnetarMarketModule._withdrawToChain()` allows withdrawal to another chain, it requires `refundAddress` to be specified so that excess gas can be refunded to an address on the source chain. However, `refundAddress` is incorrectly set as it does not handle all cases.

```
// build LZ params
        bytes memory _adapterParams;
        ICommonOFT.LzCallParams memory callParams = ICommonOFT.LzCallParams({
            refundAddress: msg.value == gas ? refundAddress : payable
            //(this),  //@audit this does not handle all cases
            zroPaymentAddress: address(0),
            adapterParams: ISendFrom(address(asset)).useCustomAdapterParams()
                ? adapterParams
                : _adapterParams
        });
```

I have determined 4 different cases of how `_withdrawToChain()` can be called, and the following shows that cases 2/3/4 are incorrect.

Case 1: Direct call

`gas == msg.value` and `refundAddress` will be from parameter This is correct, assuming no user error.

```
EOA -> MagnetarV2.withdrawToChain()
        -> MagnetarMarketModule._withdrawToChain() // delegatecall
```

Case 2: Cross-chain direct call

`gas == msg.value` and `refundAddress` will be `payable(to)` due to BaseTOFTMarketDestinationModule.sol#L247 This is incorrect as `refundAddress` should a user provided address on the source chain.

```
LZ (dest) -> BaseTOFT._nonblockingLzReceive()
                -> BaseTOFTMarketDestinationModule.remove() //delegatecall
                        -> MagnetarV2.withdrawToChain()
                                -> MagnetarMarketModule._withdrawToChain() // delegatecall
```

Case 3: Cross-chain indirect call

`gas == msg.value` and `refundAddress` will be `payable(this)` due to MagnetarMarketModule.sol#L858 This is incorrect as `refundAddress` should be a user provided address on the source chain.

```
LZ (dest) -> BaseTOFT._nonblockingLzReceive()
                -> BaseTOFTMarketDestinationModule.borrowInternal() //delegatecall
                        -> MagnetarV2.depositAddCollateralAndBorrowFromMarket()
                                -> MagnetarMarketModule.depositAddCollateralAndBorrowFromMarket
    //() // delegatecall
                                        -> MagnetarMarketModule._withdraw()
                                                -> MagnetarMarketModule._withdrawToChain()
```

Case: 4. Multicall using burst()

`gas == _action.value` and `refundAddress` will be `payable(this)` due to MagnetarMarketModule.sol#L800 This is incorrect as `refundAddress` should be `msg.sender`.

```
EOA -> MagnetarV2.burst()
        -> MagnetarV2.depositAddCollateralAndBorrowFromMarket()
                -> MagnetarMarketModule.depositAddCollateralAndBorrowFromMarket
  //() // delegatecall
                        -> MagnetarMarketModule._withdraw()
                                -> MagnetarMarketModule._withdrawToChain()
```

Recommendations

Allow `refundAddress` to be passed in as a parameter from the first `MagnetarV2` function. As there are multiple cases as mentioned, this makes it easier to determine the correct refundAddress, instead of inferring it via `gas` or msg.value.

For cross chain calls (case 2 & 3), the `refundAddress` should actually be a user provided address on the source chain, and not `payable(to)`. That is because when withdrawing to a different chain, `payable(to)` will be the address on dest chain, which may not be a user controlled address on the source chain (in the case of a multisig wallet).

[M-10] `TapOracle` TWAP duration for Uniswap Oracle should be at least 30 mins

Severity

Impact: High, will cost loss of funds for oTAP options exercise

Likelihood: Low, this could occur when liquidity is not spread over a wide range (e.g. during protocol launch)

Description

`TapOracle` uses a single price feed based on Uniswap V3 TWAP oracle for TAP/USDC pool.

However, the deploy script for `TapOracle` indicates a TWAP duration of 600 secs (10mins), which is lower than typical 1,800 secs (30mins) that is used by Euler[1] and Uniswap[2] in their studies. This puts `TapOracle` at a higher risk of price manipulation when liquidity of the TAP/USDC pool is not spread over a wide range. This could occur during protocol launch where the Uniswap pool has limited liquidity.

One may argue that such price manipulation is risky for the attacker, as the attacker has to use their own capital (instead of flash loan) to keep the price manipulated for more than a block, making them vulnerable to arbitrage. But that is not a total deterrence as shown in Rari's Fuse hack[3], where the attacker risked their capital and waited for multiple blocks. The root cause of that hack was due to price manipulation of the Uniswap V3 TWAP oracle,

which had a TWAP duration of 600 secs and the Uniswap pool did not have liquidity over a wide range.

```
const args: Parameters<Seer__factory['deploy']> = [
        'TAP/USDC', // Name
        'TAP/USDC', // Symbol
        18, // Decimals
        [
            ARGS_CONFIG[chainID].TAP_ORACLE.TAP_ADDRESS, // TAP
            ARGS_CONFIG[chainID].MISC.USDC_ADDRESS, // USDC
        ],
        [
            ARGS_CONFIG[chainID].TAP_ORACLE.TAP_USDC_LP_ADDRESS, /// LP TAP/USDC
        ],
        [1], // Multiply/divide Uni
        600, // @audit Uniswap V3 TWAP duration 600 seconds is lower than
        // typical 30 mins
        10, // Observation length that each Uni pool should have
        0, // Whether we need to use the last Chainlink oracle to convert to
        // another
        // CL path
        [],
        [], // Multiply/divide CL
        86400, // CL period before stale, 1 day
        [deployer.address], // Owner
        hre.ethers.utils.formatBytes32String('TAP/USDC'), // Description,
        ARGS_CONFIG[chainID].MISC.CL_SEQUENCER, // CL Sequencer
        deployer.address, // Owner
    ];
```

References

[1] https://docs.euler.finance/euler-protocol/eulers-default-parameters#twap-length

[2] https://blog.uniswap.org/uniswap-v3-oracles

[3] https://cmichel.io/replaying-ethereum-hacks-rari-fuse-vusd-price-manipulation/

Recommendations

Set the `TapOracle` Uniswap V3 Oracle TWAP duration to be at least 30 mins. Note that it is also important to ensure the TAP/USDC pool liquidity is spread over a wide range to increase the attack cost.

[M-11] Missing gas forwarding in cross-chain call

Severity

Impact: Medium, broken functionality of an important function

Likelihood: Medium, `TOFT_REMOVE_AND_REPAY` operation always reverts when called

Description

The operation `TOFT_REMOVE_AND_REPAY` is used to exit a position and then remove collateral from a market. The issue is that the function being called sends out a LayerZero call, but no gas is forwarded to it.

The function call can be found in `MagnetarV2.sol` contract under the action id `TOFT_REMOVE_AND_REPAY` as shown below.

```
if (_action.id == TOFT_REMOVE_AND_REPAY) {
                HelperTOFTRemoveAndRepayAsset memory data = abi.decode(
                    _action.call[4:],
                    (HelperTOFTRemoveAndRepayAsset)
                );

                _checkSender(data.from);
                IUSDOBase(_action.target).removeAsset(
                    data.from,
                    data.to,
                    data.lzDstChainId,
                    data.zroPaymentAddress,
                    data.adapterParams,
                    data.externalData,
                    data.removeAndRepayData,
                    data.approvals,
                    data.revokes
                );
```

Since no gas is forwarded to the external call, the external call will have `msg.value` of 0. However if we check the `removeAsset` function in BaseUSDO.sol, we see a subsequent layerzero call via the USDOMarketModule.

```
bytes memory lzPayload = abi.encode(
            PT_MARKET_REMOVE_ASSET,
            to,
            externalData,
            removeAndRepayData,
            approvals,
            revokes,
            airdropAmount
        );

        _checkAdapterParams(
            lzDstChainId,
            PT_MARKET_REMOVE_ASSET,
            adapterParams,
            NO_EXTRA_GAS
        );

        _lzSend(
            lzDstChainId,
            lzPayload,
            payable(from),
            zroPaymentAddress,
            adapterParams,
            msg.value
        );
```

The issue here is that `msg.value` is 0, hence no gas will be sent to the layerzero endpoint, failing the cross-chain call.

Recommendations

Modify the call in magnetar to forward the gas.

```
IUSDOBase(_action.target).removeAsset{value: _action.value}(...)
```

[M-12] Not maximizing `Glp` in `GlpOracle` miscalculates `_minGlp` when adding liquidity

Severity

Impact: Medium. The rounding will be incorrect affecting the price.

Likelihood: High. False is hardcoded so it is very likely it will happen.

Description

Currently, in the `GlpOracle` contract, there is a function that fetches Glp price from the Glp Manager:

```
function _get() internal view returns (uint256) {
        return glpManager.getPrice(false);
    }
```

The price maximation is set to false `glpManager.getPrice(false);` . This price is then fetched from the `GlpStrategy` in the Yieldbox module. In this case, the price has a part on the amountOut or the `_minGlp` that we expect when adding liquidity to GMX.

```
(success, glpPrice) = wethGlpOracle.get(wethGlpOracleData);
            if (!success) revert Failed();
            uint256 amountInGlp = (wethAmount * glpPrice) / 1e18;
            amountInGlp = amountInGlp - (amountInGlp * _slippage) / 10_000;
```

Given that the price maximation is set to `false`, it will return a slightly smaller price, which will set the slippage as lower than if it were set with `maximize = true`.

This will allow the `GlpManager` to mint less Glp to the user without reverting as `_minGlp` with maximize set to `false`:

`require(mintAmount >= _minGlp, "GlpManager: insufficient GLP output");`

Recommendations

```
function _get() internal view returns (uint256) {
-       return glpManager.getPrice(false);
+       return glpManager.getPrice(true);
    }
```

[M-13] Any tokens or eth in Magnetar can be drained

Severity

Impact: High, funds can be stolen, approvals can be given out to random addresses

Likelihood: Low, Magnetar is not designed to hold user funds

Description

The Magnetar contract issues approvals to certain contracts such as the YieldBox contract and the Singularity and BigBang market contracts to do token transfers. The issue is that it takes all these addresses as user inputs. Thus a malicious user can exploit this to give approvals to malicious contracts which can then drain any tokens present in the Magnetar contract.

While the Magnetar contract itself is not designed to hold any user funds, it does have a recovery function for ETH, and might hold some eth for gas for LZ calls. These funds can also be stolen.

As an example, lets look at the `withdrawToChain` function in `MagnetarMarketModule.sol` contract.

```
function withdrawToChain(
        IYieldBoxBase yieldBox,
        address from,
        uint256 assetId,
        uint16 dstChainId,
        bytes32 receiver,
        uint256 amount,
        bytes memory adapterParams,
        address payable refundAddress,
        uint256 gas,
        bool unwrap
    )
```

The address for `yieldBox` is taken as an input from the user and never verified. Then in the `_withdrawToChain` function, this address is interacted with. Lets assume `dstChainId` passed is non-zero. Then, the contract calls this malicious `yieldbox` contract to get another address, asset.

```
(, address asset, , ) = yieldBox.assets(assetId);
```

This `asset` address is also malicious user input. Then, if `unwrap` is set to `true`, the function calls this `asset` address with some eth, the amount of which is dictated by `gas`, another user input.

```
if (unwrap) {
            ICommonData.IApproval[]
                memory approvals = new ICommonData.IApproval[](0);
            ITapiocaOFT(address(asset)).triggerSendFromWithParams{value: gas}(
                address(this),
                dstChainId,
                receiver,
                amount,
                callParams,
                true,
                approvals,
                approvals
            );
```

Since `asset` is also a malicious address, it can just receive this free eth from the contract and finish this contract call. Thus the attacker has gained free eth from the contract.

Similarly, any other ERC20 tokens can also be drained for example in the `_depositAddCollateralAndBorrowFromMarket` function, where the contract does calls to the `market` value passed in by the user as input.

```
IYieldBoxBase yieldBox = IYieldBoxBase(market.yieldBox());

        uint256 collateralId = market.collateralId();
        (, address collateralAddress, , ) = yieldBox.assets(collateralId);
```

The malicious `market` contract can return more malicious addresses, and then `extractTokens` is called to do token transfers which the attacker can use to take out any stored tokens in Magnetar. Approvals are also given to `yieldbox` which can be a malicious address.

```
if (deposit) {
            // transfers tokens from sender or from the user to this contract
            collateralAmount = _extractTokens(
                extractFromSender ? msg.sender : user,
                collateralAddress,
                collateralAmount
);
IERC20(collateralAddress).approve(address(yieldBox), 0);
            IERC20(collateralAddress).approve(
                address(yieldBox),
                collateralAmount
            );
```

So using a variety of functions, any funds left in Magnetar can be drained.

Recommendations

If no funds at all are to be stored in Magnetar, this can be safely ignored. Otherwise, the addresses for `yieldbox` and `markets` need to be checked against a whitelist of contracts. Such a pattern already exists using the `cluster` variable in the `_mintFromBBAndLendOnSGL` function.

```
if (externalContracts.bigBang != address(0)) {
            //@audit cluster check useless since passed in by the user
            if (
                !_cluster.isWhitelisted(
                    _cluster.lzChainId(),
                    externalContracts.bigBang
                )
            ) revert NotAuthorized();
        }
```

[M-14] Magnetar `_depositRepayAndRemoveCollateralFromMarket` can fail due to rounding errors

Severity

Impact: Medium, can cause function to revert

Likelihood: Medium, happens when user tries to remove collateral and withdraw from yieldbox

Description

If `collateralAmount` is set to be more than 0 while calling the `_depositRepayAndRemoveCollateralFromMarket` function, the following snippet is executed.

```
if (collateralAmount > 0) {
            address collateralWithdrawReceiver = withdrawCollateralParams
                .withdraw
                ? address(this)
                : user;
            uint256 collateralShare = yieldBox.toShare(
                marketInterface.collateralId(),
                collateralAmount,
                false
            );
            marketInterface.removeCollateral(
                user,
                collateralWithdrawReceiver,
                collateralShare
            );

            //withdraw
            if (withdrawCollateralParams.withdraw) {
                _withdrawToChain(
                    yieldBox,
                    collateralWithdrawReceiver,
                    marketInterface.collateralId(),
                    withdrawCollateralParams.withdrawLzChainId,
                    LzLib.addressToBytes32(user),
                    collateralAmount,
                    withdrawCollateralParams.withdrawAdapterParams,
                    valueAmount > 0 ? payable(msg.sender) : payable(this),
                    valueAmount,
                    withdrawCollateralParams.unwrap
                );
            }
        }
```

In the first bit, `collateralShare` is calculated from the yieldBox rebase calculations. since the last parameter is false, the answer is rounded down. Lets assume `collateralAmount` is set to 100, and the `collateralShare` calculated is between 49 and 50, and is set as 49 due to the rounding down.

Later in the `_withdrawToChain` call, the contract tries to withdraw the full `collateralAmount` amount of tokens. Here the yieldBox is called again to take those tokens out of it.

```
yieldBox.withdraw(
                assetId,
                from,
                LzLib.bytes32ToAddress(receiver),
                amount,
                0
            );
```

But in this call, due to it being a withdraw, the amount of shares to be deducted is rounded up. So now for withdrawing the same 100 units of `collateralAmount`, the amount of shares to be burnt is calculated as 50 due to the rounding up. Since the contract has withdrawn only 49 shares but the yieldBox is trying to burn 50, this function will revert.

This is similar to the M-02 report showing a similar rounding error forcing a revert but in a different scenario.

Recommendations

Calculate `collateralShare` by rounding it up. This will ensure enough shares are present to burn during withdrawal.

```
uint256 collateralShare = yieldBox.toShare(
                marketInterface.collateralId(),
                collateralAmount,
                true
            );
```

[M-15] Using only DEX swappers is suboptimal

Severity

Impact: Medium, swaps without aggregator might have a high price impact for large amounts.

Likelihood: Medium. Depending on how much amount it is being swapped this can be a frequent problem.

Description

Currently, most of the actions that need tokens to be swapped across Tapioca's codebases use the swappers, which currently, they are 3.

Each swapper uses a different DEX, UniV2, Univ3 and Curve DeFi pools. This is not the ideal scenario for most cases as when you are swapping you are trying to maximize the amountOut that you get in exchange for the tokens you swapped. To accomplish this and get better rates for your swaps, at least one aggregator should be added to the list of swappers.

Recommendations

Add at least one aggregator to the list of swappers. 1inch is my preferred one, but you could also go with 0x.

[M-16] Incorrect order when calculating the transferred amount to Stargate makes the

`participate()` function unusable

Severity

Impact: Medium, as the functionality will always revert

Likelihood: Medium, as it only occurs when calling `participate`

Description

The participate() function in Stargate helper contract will not work.

The cause of this, is the way on how you Tapioca is calculating the amount transferred to Stargate to refund the dust amounts that Stargate returns:

```
uint256 transferred = balanceAfter - balanceBefore;
```

Let's put an example without decimals for simplicity.

- I, as the msg.sender, transfer 50 tokens: `balanceBefore = 50`
- Those 50 tokens are then sent to the pool. Let's say Stargate returns 1 token as dust.
- Then it will be: `uint256 transferred = 1 - 50;`

therefore the calculation will always revert.

Recommendations

```
uint256 balanceAfter = IERC20(stargateData.srcToken).balanceOf(
            address(this)
        );
+        uint256 transferred = balanceBefore  - balanceAfter;
-        uint256 transferred = balanceAfter - balanceBefore;
```

8.4. Low Findings

[L-01] Redeem functions and `retryRevert()` in `StargateLbpHelper` are unnecessary

There are redeem functions in `StargateLbpHelper` to remove liquidity from stargate but there are no function to add liquidity, making them unnecessary.

Also, `retryRevert()` is not required as anyone can directly call the existing `retryRevert()` in `StargateLbpHelper` to retry the stargate swap.

To resolve this, remove redeem functions and , `retryRevert()`.

[L-02] `getOutputAmount()` should not be used in state-changing functions

It is assumed that the intention of `getOutputAmount()` in both `UniswapV2Swapper` and `UniswapV3Swapper` is for view functions to build the swap parameters at the frontend. That is because it provides either a spot price or a low TWAP duration price. Both of these should not be used in state-changing functions, as the prices can be manipulated and affect the state-changing functions.

However, it is used in state-changing functions `activateBid()` and `removeBid()` within `LiquidationQueue`, which could allow price manipulation to affect them.

Set this as Low as `LiquidationQueue` is out of scope for the audit and I assume the `getOutputAmount()` is designed only for view functions.

To resolve this, do not use `getOutputAmount()` in state-changing functions.

[L-03] Incorrect `interfaceId` check in `_withdrawToChain()`

`MagnetarMarketModule._withdrawtoChain()` will check if the asset supports cross chain transfer using `supportsInterface()` as follows:

```
try
            IERC165(address(asset)).supportsInterface(
                type(ISendFrom).interfaceId
            )
        {} catch {
            yieldBox.withdraw(
                assetId,
                from,
                LzLib.bytes32ToAddress(receiver),
                amount,
                0
            );
            return;
        }
```

However, LayerZero `BaseOFTV2sol` that supports `sendFrom()` is actually returning `type(IOFTV2).interfaceId` as such:

```
function supportsInterface(bytes4 interfaceId) public view virtual override
      (ERC165, IERC165) returns (bool) {
        return interfaceId == type
          (IOFTV2).interfaceId || super.supportsInterface(interfaceId);
    }
```

While LayerZero `OFTCore.sol` returns `type(IOFTCore).interfaceId`

```
function supportsInterface(bytes4 interfaceId) public view virtual override
      (ERC165, IERC165) returns (bool) {
        return interfaceId == type
          (IOFTCore).interfaceId || super.supportsInterface(interfaceId);
    }
```

And `TapiocaOFT` does not return an interfaceId of its functions - `triggerSendFromWithParams()`, and only return OFTV2 `interfaceId` based on inheritance.

There are 3 issues with the check,

1. The `supportInterface()` check will actually pass even when the asset does not support `sendFrom()` as it is checking using the wrong `interfaceId`.

2. The `supportInterface()` check will also pass as long as the `supportInterface()` check does not revert, even if the asset does not support `sendFrom()`. Thats because there is no handling of the return value of `supportInterface()`, which will return false in this case.

3. When `unwrap == true`, the check will also pass even if the asset is not a `TapiocaOFT` that supports `triggerSendFromWithParams()`.

It is possible that `_withdrawtoChain()` will be called at the destination chain during a cross chain call. When that occurs, this issue will cause the tokens to be stuck in `Magnetar` contract if it cannot withdraw them on another chain. However, I am categorizing it as a Low, as this only happens on user error, when trying to withdraw an unsupported asset on another chain.

This issue can be fixed as follows:

1. For use of `sendFrom()`, check that the asset supports either `type(IOFTCore).interfaceId` or `type(IOFTV2).interfaceId`.
2. For use of `triggerSendFromWithParams()`, check that the asset supports TapiocaOFT `interfaceId`, which has to be added.
3. For the above, check the return value of `supportInterface()` and do a local chain withdrawal when it returns false. Note: the try/catch can remain, to handle any assets that did not implement IERC165.

Alternatively, an easier fix is to perform a try/catch on `triggerSendFromWithParams()` and `sendFrom()` to handle the scenario where the withdrawn asset does not support cross chain transfer.

[L-04] Protocol is using old version of OpenZeppelin contracts

The version currently used is 4.5.0 which is quite outdated. All versions of OpenZeppelin until 4.9.3 have associated bugs with them. While the scope is not directly impacted by the issues reported in 4.5.0 it is recommended to use the earliest version in which a bug has not been found. Currently this is version 4.9.3. I would like to note that one of the high severity risks from version 4.5.0 is associated with `ERC165Checker.supportsInterface`. This contract is inherited from `ProxyONFT721` and `ProxyONFT1155`. They are not in scope, so the severity of this finding against the contracts in scope is Low. However in OZ v4.5.0 `ERC165Checker.supportsInterface` does not revert under any circumstances and always return true. More information here. Consider upgrading to v4.9.3.

[L-05] `answeredInRound` is deprecated

In `ChainlinkUtils.sol` the `_readChainlinkBase()` function uses the deprecated variable when checking if `roundId` is greater than `answeredInRound`.

Chainlink has recently added that `answeredInRound` is now deprecated, so you should be careful when using it.

- `answeredInRound`: Deprecated - Previously used when answers could take multiple rounds to be computed.

Reference

[L-06] `zroPaymentAddress` should not be hardcoded to `address(0)`

The LayerZero integration checklist says:

"Do not hardcode address zero (`address(0)`) as `zroPaymentAddress` when estimating fees and sending messages. Pass it as a parameter instead."

But in `MagnetarMarketModule.sol`, `zroPaymentAddress` is hardcoded to `address(0)`:

```
// build LZ params
        bytes memory _adapterParams;
        ICommonOFT.LzCallParams memory callParams = ICommonOFT.LzCallParams({
            refundAddress: msg.value == gas ? refundAddress : payable(this),
            zroPaymentAddress: address(0), //@audit should not be hardcoded
            adapterParams: ISendFrom(address(asset)).useCustomAdapterParams()
                ? adapterParams
                : _adapterParams
        });
```

Remove the hardcoded `address(0)` and pass it as a parameter instead.

[L-07] `StargateLbpHelper.participate()` could leave dust amount in contract

In `StargateLbpHelper.participate()`, `stargateData.amount` of `srcToken` is transferred into the contract and then sent to another chain via Stargate `router.swap()`.

However, Stargate `router.swap()` may send less than the entire `stargateData.amount` due to a conversion that will leave dust amount in the contract. This will occur for tokens that has different decimals on different chains, where Stargate router will convert from a local decimals (on source chain) to a shared decimals (normalized for all chains).

For example, Stargate LUSD pool has a local decimals of 18 for Optimism and shared decimals of 6. So the conversion (from local decimals to shared decimals and then back to local decimals) will remove the dust amount that is not represented in shared decimals, which is lower than local decimals.

```
function participate(
        StargateData calldata stargateData,
        ParticipateData calldata lbpData
    ) external payable nonReentrant {
        IERC20 erc20 = IERC20(stargateData.srcToken);

        // retrieve source token from sender
        erc20.safeTransferFrom(msg.sender, address(this), stargateData.amount);

        // compute min amount to be received on destination
        uint256 amountWithSlippage = stargateData.amount -
            ((stargateData.amount * stargateData.slippage) /
                SLIPPAGE_PRECISION);

        // approve token for Stargate router
        erc20.safeApprove(address(router), stargateData.amount);

        //@audit this could leave dust amount due to a conversion within swap()
        // send over to another layer using the Stargate router
        router.swap{value: msg.value}(
            stargateData.dstChainId,
            stargateData.srcPoolId,
            stargateData.dstPoolId,
            payable(msg.sender), //refund address
            stargateData.amount,
            amountWithSlippage,
            IStargateRouterBase.lzTxObj({
                dstGasForCall: 0,
                dstNativeAmount: 0,
                dstNativeAddr: "0x0"
            }),
            abi.encodePacked
            //(msg.sender), // StargateLbpHelper.sol destination address
            abi.encode(lbpData, msg.sender)
        );
    }
```

```
function swap(
        uint16 _dstChainId,
        uint256 _srcPoolId,
        uint256 _dstPoolId,
        address payable _refundAddress,
        uint256 _amountLD,
        uint256 _minAmountLD,
        lzTxObj memory _lzTxParams,
        bytes calldata _to,
        bytes calldata _payload
    ) external payable override nonReentrant {
        require(_amountLD > 0, "Stargate: cannot swap 0");
        require(_refundAddress != address
          (0x0), "Stargate: _refundAddress cannot be 0x0");
        Pool.SwapObj memory s;
        Pool.CreditObj memory c;
        {
            Pool pool = _getPool(_srcPoolId);
            {
                uint256 convertRate = pool.convertRate();
                //@audit this converts from local decimals to shared decimals
                // and then back to local decimals.
                         it will remove dust amount that is not represented
                // in shared decimals
                _amountLD = _amountLD.div(convertRate).mul(convertRate);
            }

            s = pool.swap(
              _dstChainId,
              _dstPoolId,
              msg.sender,
              _amountLD,
              _minAmountLD,
              true
            );
            _safeTransferFrom(pool.token(), msg.sender, address
              (pool), _amountLD);
            c = pool.sendCredits(_dstChainId, _dstPoolId);
        }
        bridge.swap{value: msg.value}(
          _dstChainId,
          _srcPoolId,
          _dstPoolId,
          _refundAddress,
          c,
          s,
          _lzTxParams,
          _to,
          _payload
        );
    }
```

This could be resolved by checking the actual amount that was transferred by `router.swap` and use it to determine the leftover dust that will be refunded to the caller.

[L-08] Un-used `ParticipateData.poolId` for `StargateLbpHelper`

`StargateLbpHelper.participate()` takes in a `ParticipateData` struct parameter, which has a `poolId`.

However, `ParticipateData.poolId` is not used for the destination chain `sgReceive()` execution. This will slightly increases the LayerZero fee as it is based on the payload size.

```
struct ParticipateData {
        address assetIn;
        address assetOut;
        uint256 poolId; //@audit this is not used for destination chain
        // execution
        uint256 deadline;
        uint256 minAmountOut;
    }
```

This can be resolved by removing `poolId` from the `ParticipateData` struct.

[L-09] Useless payload check

The payload that is being sent to stargate includes the following data: `abi.encode(lbpData, stargateData.receiver)`

but at the `sgReceive()` function there is this check: `if (payload.length <= 40) return;`

I realize this check is done because it is in stargate docs, but they are using a different payload that includes 2 addresses + other data, that is why the 40 bytes length check exists, to make sure the addresses are in fact 20 bytes.

Given that you have to send the `ParticipateData` struct and the components of the payload are not denominated in bytes, the check is useless.

Remove the following check from the `sgReceive()` function.

```
- if (payload.length <= 40) return;
```

[L-10] Do not hardcode `_payInZRO` to false when estimating layerZero fees

The LayerZero integration checklist says:

> "Do not hardcode useZro to false when estimating fees and sending messages. Pass it as a parameter instead."

But in `StargateLbpHelper.sol`, `_payInZRO` is hardcoded to `false`:

```
endpoint.estimateFees(
                _dstChainId,
                address(bridge),
                payload,
                false,
                _txParamBuilder(_dstChainId, _functionType, _lzTxParams)
            );
```

Remove the hardcoded `false` and pass it as a parameter instead, even if the intended usage is to always use `false`

```
endpoint.estimateFees(
                _dstChainId,
                address(bridge),
                payload,
+                _payInZRO,
-                false,
                _txParamBuilder(_dstChainId, _functionType, _lzTxParams)
            );
```