Tapioca DAO - Magnetar Security Review

Conducted by: 0xadrii

April 12, 2024 - April 21, 2024

Contents

1. Introduction

1.1 About 0xadrii

I specialize in conducting smart contract audits as an independent security researcher. My expertise includes a proven track record in public audit contests with numerous top 3 finishes and bug bounties, along with extensive experience in evaluating complex and high-profile protocols. You can find my previous work **here** or reach out on Twitter at **@0xadrii**.

1.2 About Tapioca DAO

TapiocaDAO is a *decentralized autonomous organization* (*DAO*) which created a decentralized Omnichain stablecoin ecosystem, comprised of multiple sub-protocols, which includes; **Singularity**, the first-ever Omnichain isolated money market, **Big Bang**, an Omnichain CDP Stablecoin Creation Engine, **Yieldbox**, the most powerful token vault ever created, **tOFT** (Tapioca Omnichain Wrapper[s]) which transforms any fragmented asset into a unified Omnichain asset, **twAML**, an economic incentive consensus mechanism, and **Pearlnet**, **the self-sovereign Omnichain verifier network.

1.3 Disclaimer

This report presents an analysis conducted within specific parameters and timeframe, relying on provided materials and documentation. It **does not** encompass all possible vulnerabilities and should not be considered exhaustive. The review and accompanying report are provided on an 'as-is' and 'as-available' basis, without any express or implied warranties. Additionally, this report does not endorse any particular project or team, nor does it guarantee the absolute security of the project.

2. Risk classification

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

2.1 Impact

- High: Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality.
- Medium: Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality.
- Low: Funds are **not** at risk.

2.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized.
- Medium - only conditionally possible or incentivized, but still relatively likely.
- Low - really improbable, requires little-to-no incentive.

2.3 Action required for severity levels

- High: must fix as soon as possible.
- Medium: should fix.
- Low: could fix.

3 Executive summary

Over the course of ten days in total, Tapioca DAO engaged with 0xadrii to review Magnetar. In this period of time, a total of 21 were found.

3.1 Overview

Project	**Tapioca Periphery - Magnetar**
Repository	https://github.com/Tapioca-DAO/tapioca-periph
Commit	https://github.com/Tapioca-DAO/tapioca-periph/commit/44c5845cc68a5b87042764d471952b898562cc12
Audit timeline	April 12, 2024 - April 21, 2024

3.2 Issues found

Severity	**# of issues**
High	7
Medium	3
Low	4
Gas optimization	5
Informational	2
Total Issues	21

4 Findings

High

[H-01] - Wrong data decoding makes it impossible to unwrap OFT's via Magnetar

Context: Magnetar.sol#L272

Severity: High

OFT operations in Magnetar are handled via the internal `_processOFTOperation` function. Prior to actually executing the OFT call, a validation process takes place, where the encoded data for the function call is validated:

```
function _processOFTOperation(
        address _target,
        bytes calldata _actionCalldata,
        uint256 _actionValue,
        bool _allowFailure
    ) private {
        ...

        if (funcSig == ITOFT.unwrap.selector) {
            (, uint256 _amount) = abi.decode(_actionCalldata[4:36], (ad
            // IERC20(_target).safeTransferFrom(msg.sender, address(thi
            {
                bool isErr = pearlmit.transferFromERC20(msg.sender, add
                if (isErr) revert Magnetar_PearlmitTransferFailed();
            }
        }
```

As shown in the code snippet, for calls that aim at unwrapping TOFTs, the `_actionCalldata` parameter is expected to be decoded as two variables of type `address` and `uint256`. The problem is that the size to be decoded is only of 32 bytes (from byte 4 to byte 36). This will make all unwrapping operations fail, because data from an `address` and `uint256` can't possible be stored inside 32 bytes only.

Recommendation

Change the `_actionCalldata` decoding size to 64 bytes:

```
function _processOFTOperation(
        address _target,
        bytes calldata _actionCalldata,
        uint256 _actionValue,
        bool _allowFailure
    ) private {
        ...

        if (funcSig == ITOFT.unwrap.selector) {
-            (, uint256 _amount) = abi.decode(_actionCalldata[4:36], (a
+            (, uint256 _amount) = abi.decode(_actionCalldata[4:68], (a
            // IERC20(_target).safeTransferFrom(msg.sender, address(thi
            {
                bool isErr = pearlmit.transferFromERC20(msg.sender, add
                if (isErr) revert Magnetar_PearlmitTransferFailed();
            }
        }
```

[H-02] - Transferring wrong asset makes it impossible to participate in twAML voting

Context: Magnetar.sol#L272

Severity: High

Magnetar's `_processTapLockOperation` internal function allows to perform several operations related to Tap tokens. One of these operations includes participating in twAML voting to mint an oTAP position. In order to do so, TapiocaOptionBroker's `participate` function will be called. This process should perform the following steps:

1. Transfer user's tOLP tokens to Magnetar
2. Approve and execute the `participate` call
3. After locking the tOLP, transfer the minted oTAP position back to the user.

However, as shown in the following snippet, the third step is wrong. Instead of transferring back the the user the minted oTAP token, Magnetar will try to transfer a `tOLP` instead:

```
function _processTapLockOperation(
        address _target,
        bytes calldata _actionCalldata,
        uint256 _actionValue,
        bool _allowFailure
    ) private {

    ...

      if (funcSig == ITapiocaOptionBroker.participate.selector) {
            (uint256 tokenId) = abi.decode(_actionCalldata[4:], (uint25
            address tOLP = ITapiocaOptionBroker(_target).tOLP();

            {
                bool isErr = pearlmit.transferFromERC721(msg.sender, ad
                if (isErr) {
                    revert Magnetar_PearlmitTransferFailed();
                }
            }

            pearlmit.approve(tOLP, tokenId, _target, 1, (block.timestam

            (bytes memory tokenIdData) = _executeCall(_target, _actionC
            ITapiocaOptionLiquidityProvision(tOLP).safeTransferFrom(
                address(this), msg.sender, abi.decode(tokenIdData, (uin
            );

            return;
        }

       ...

}
```

This will make participating in twAML voting from Magnetar impossible, because the final `tOLP` transfer will always fail (instead, the `oTAP` token should be transferred).

Recommendation

Change the token to be transferred back to the user in the final step from the `tOLP` to the newly minted `oTAP` :

```
function _processTapLockOperation(
        address _target,
        bytes calldata _actionCalldata,
        uint256 _actionValue,
        bool _allowFailure
    ) private {

    ...

      if (funcSig == ITapiocaOptionBroker.participate.selector) {
            (uint256 tokenId) = abi.decode(_actionCalldata[4:], (uint25
            address tOLP = ITapiocaOptionBroker(_target).tOLP();

            {
                bool isErr = pearlmit.transferFromERC721(msg.sender, ad
                if (isErr) {
                    revert Magnetar_PearlmitTransferFailed();
                }
            }

            pearlmit.approve(tOLP, tokenId, _target, 1, (block.timestam

            (bytes memory tokenIdData) = _executeCall(_target, _actionC

+            address oTAP = ITapiocaOptionBroker(_target).oTAP();
-            ITapiocaOptionLiquidityProvision(tOLP).safeTransferFrom(
+            ITapiocaOptionLiquidityProvision(oTAP).safeTransferFrom(
                address(this), msg.sender, abi.decode(tokenIdData, (uin
            );

            return;
        }

       ...

}
```

[H-03] - Using wrong interface makes it impossible to call `TapiocaOptionLiquidityProvision.unlock` and `twTap.exitPosition`

Context: Magnetar.sol#L445-L446

Severity: High

Magnetar's `_processTapUnlockOperation` follows the same approach to calling Tapioca core's functions. Users will encode the function to be called together with a target, and after comparing the function signature, the function will be forwarded to the target. In the specific case of tap unlock operations, three possible functions can be called: ITwTap.exitPosition, ITapiocaOptionBroker.exitPosition and ITapiocaOptionLiquidityProvision:

```
function _processTapUnlockOperation(
        address _target,
        bytes calldata _actionCalldata,
        uint256 _actionValue,
        bool _allowFailure
    ) private {
        ...

        bytes4 funcSig = bytes4(_actionCalldata[:4]);

        ...

        // Token is sent to the owner after execute
        if (
            funcSig == ITwTap.exitPosition.selector ||
            funcSig == ITapiocaOptionBroker.exitPosition.selector ||
            funcSig == ITapiocaOptionLiquidityProvision.unlock.selector
        ) {
            _executeCall(_target, _actionCalldata, _actionValue, _allow
            return;
        }
        revert Magnetar_ActionNotValid(MagnetarAction.Market, _actionCa
    }
```

As the snippet shows, the function signature will be compared with the selectors given by the `ITwTap`, `ITapiocaOptionBroker` and `ITapiocaOptionLiquidityProvision` interfaces. The problem is that the `exitPosition` and `unlock` selectors given from the interfaces don't match the current code found in Tapioca's core contracts. As we can see, the functions found in the interfaces are:

```
interface ITwTap {
    ...
    function exitPosition(uint256 _tokenId) external returns (uint256 t
    ...
}

interface ITapiocaOptionLiquidityProvision is IERC721 {
    ...

    function unlock(uint256 tokenId, address singularity) external retu
}
```

However, the actual functions in Tapioca core are:

```
// twTAP.sol

function exitPosition(uint256 _tokenId, address _to)
        external
        nonReentrant
        whenNotPaused
        returns (uint256 tapAmount_)
    {
        ...
    }

// TapiocaOptionLiquidityProvision.sol
function unlock(uint256 _tokenId, IERC20 _singularity, address _to) ext
    ...
}
```

As we can see, the function signatures from the interface don't match the actual function definitions in the contracts. This issue arises from a change in https://github.com/Tapioca-DAO/tap-token/commit/9f784552a9442e3486e1791a1cfea9e594473c78 and https://github.com/Tapioca-DAO/tap-token/commit/08be148a740909a46cba1936a08bc29d34b95441, where the contracts were changed, but not the interfaces.

Recommendation

Update the interfaces to match the core contracts function signatures.

[H-04] - Forcing users to approve Magnetar in some operations is exploitable, making attackers capable of stealing locked tokens from users

Context: Magnetar.sol#L446

Severity: High

Some flows require the user to first approve Magnetar so that tokens can be transferred later by Magnetar when triggering a call. This is because some core functions fetch tokens directly from `msg.sender`, which will be Magnetar. In some situations, Tapioca core also allows to trigger functions passing arbitrary `to` parameters (which will be the address that will receive the tokens), enabling functions to be triggered on behalf of other users. This type of function flows shouldn't be performed in Magnetar, given that the initial user approval can be exploited by malicious users to steal the user's tokens.

Let's take the `tOLP` unlock flow as an example. When calling `unlock` in the TapiocaOptionLiquidityProvision contract, the process includes the following steps:

1. User approves Magnetar in the `TapicoaOptionLiquidityProvision` contract so that the token ID can be burnt by Magnetar on behalf of the user (this is because Magnetar will be `msg.sender` from the TapiocaOptionLiquidityProvision's perspective).

2. The approval is checked with `_isApprovedOrOwner`

3. `unlock` is called, tokens are burnt from Magnetar, and the tokens are transferred to the arbitrary `_to` address

```
function unlock(uint256 _tokenId, IERC20 _singularity, address _to) ext
        ...

              if (!_isApprovedOrOwner(msg.sender, _tokenId)) revert Not

        _burn(_tokenId);

        ...

        // Transfer the YieldBox position back to the owner
        yieldBox.transfer(address(this), _to, lockPosition.sglAssetID,
        ...

}
```

This flow is dangerous, because once the user has approved Magnetar (if not done in a batched manner), any malicious user can leverage the approval to call the `unlock` function via Magnetar with the victim's `tokenId`, burning the NFT from the victim and transferring the YieldBox assets to the attacker.

Recommendation

Generally, it is not recommended to allow these types of flows in Magnetar. One way to prevent this is to prevent aribtrary `to` addresses to be passed in Tapioca's core. This will make all tokens be transferred to Magnetar instead of the `to` address, allowing Magnetar to then transfer the tokens to the actual user calling the function.

[H-05] - Cross-chain calls can't be triggered from Magnetar

Context: MagnetarBaseModule.sol#L126

Severity: High

Magnetar includes some flows where cross-chain calls can be triggered. These cross-chains actions are executed directly in Magnetar's modules, and usually require some value to be sent in order to pay for the cross-chain call. However, Magnetar **does not send any value** when delegatecalling to its modules. This will prevent any cross-chain call from taking place. One example of this issue arises when a withdrawal to another chain is requested when interacting with the `AssetModule`. Let's examine how this flow would work:

1. Initially, the user will call Magnetar's `burst` with actionId equal to `MagnetarAction.AssetModule`. This will make Magnetar's internal `_executeModule` function be triggered, which will execute a `delegatecall` to the target. It is relevant to note that **no value is sent when the delegatecall is executed:**

```
function burst(MagnetarCall[] calldata calls) external payable {

     ...
     if (_action.id == MagnetarAction.AssetModule) {
       _executeModule(MagnetarModule.AssetModule, _action.call);
       continue; // skip the rest of the loop
   }

   ...
}

function _executeModule(MagnetarModule _module, bytes memory _data)
        bool success = true;
        address module = modules[_module];
        if (module == address(0)) revert Magnetar_ModuleNotFound(_m

        (success, returnData) = module.delegatecall(_data);
        if (!success) {
            _getRevertMsg(returnData);
        }
    }
```

2. The `AssetModule` will be delegatecalled. In this example, the user only wants to perform a collateral removal operation. This will make `_withdrawToChain` be called:

```
function depositRepayAndRemoveCollateralFromMarket(DepositRepayAndRe
        public
        payable
    {

        ...
            if (data.collateralAmount > 0) {
                ...
                if (data.withdrawCollateralParams.withdraw) {

                    ...
                    _withdrawToChain(data.withdrawCollateralParams)

                }
            }
        }
```

3. When executing `_withdrawToChain`, the user will specify a `dstEid` different than 0, which means a cross-chain withdrawal is requested. The internal `_lzWithdraw` call will then be triggered:

```
 function _withdrawToChain(MagnetarWithdrawData memory data) interna

    ...

    // perform a same chain withdrawal
    if (data.lzSendParams.sendParam.dstEid == 0) {
        _withdrawHere(_yieldBox, data.assetId, data.lzSendParams.sen
        return;
    }

    ...

    if (data.unwrap) {
           ...
    } else {
        _lzWithdraw(asset, data.lzSendParams, data.sendGas, data.sen
    }
```

4. Finally, `_lzWithdraw` will interact with `_asset` to call its `sendPacket` function, which will perform the cross-chain call. We can see how two problems will arise due to the fact that no value was sent when delegatecalling from Magnetar to the module:

    1. The `if (msg.value < prepareLzCallReturn.msgFee.nativeFee)` will always fail, because `msg.value` is always 0.
    2. The `prepareLzCallReturn.msgFee.nativeFee` specified as value in `sendPacket` can't actually be sent, because no value has been sent at all.

```
function lzWithdraw(address asset, LZSendParam memory lzSendPara
        private
    {
        PrepareLzCallReturn memory prepareLzCallReturn = prepare
```

```
if (msg.value < prepareLzCallReturn.msgFee.nativeFee) {
revert Magnetar_GasMismatch(prepareLzCallReturn.msgFee.nativeFee, msg.value);
```

```
IOftSender(_asset).sendPacket{value: prepareLzCallReturn.msgFee.nativeFee}(
prepareLzCallReturn.lzSendParam, prepareLzCallReturn.composeMsg ); }
```

Recommendation

Allow the call's value to be transferred when delegatecalling into modules. This will allow the value to be properly sent accross calls, enabling cross-chain interactions to be performed.

[H-06] - Compose messages will wrongly hardcode source chain sender as Magnetar

Context: General

Severity: High

Magnetar allows users to directly perform cross-chain calls by triggering Tapioca's OFT tokens' `sendPacket` functions. These cross-chain calls can include compose messages, which allow users to chain several cross-chain calls into one. In order to identify who the caller of a compose call is, Tapioca will always encode the caller address as the source chain sender. This address will be decoded and verified on the destination chain.

These type of calls, however, can't be performed from Magnetar because Tapioca's core always hardcodes the source chain sender as the `msg.sender`. Let's examine a flow where a compose message would be triggered from Magnetar:

1. User calls Magnetar's `_processOFTOperation` function with the intention to trigger a tOFT's `sendPacket` function.
2. The tOFT's `sendPacket` function is called, which will internally trigger the `TapiocaOmnichainSender` 's `sendPacket` function:

```
// TOFT.sol

function sendPacket(LZSendParam calldata _lzSendParam, bytes calldat
        public
        payable
        whenNotPaused
        returns (MessagingReceipt memory msgReceipt, OFTReceipt memo
    {
        (msgReceipt, oftReceipt) = abi.decode(
            _executeModule(
                uint8(ITOFT.Module.TOFTSender),
                abi.encodeCall(TapiocaOmnichainSender.sendPacket, (_
                false
            ),
            (MessagingReceipt, OFTReceipt)
        );
    }
```

3. `TapiocaOmnichainSender` 's `sendPacket` will trigger `_buildOFTMsgAndOptions` , which builds the cross-chain call:

```
function sendPacket(LZSendParam calldata _lzSendParam, bytes calldat
        external
        payable
        returns (MessagingReceipt memory msgReceipt, OFTReceipt memo
    {
        ...

        // @dev Builds the options and OFT message to quote in the e
        (bytes memory message, bytes memory options) =
            _buildOFTMsgAndOptions(_lzSendParam.sendParam, _lzSendPa

        // @dev Sends the message to the LayerZero endpoint and retu
        msgReceipt =
            _lzSend(_lzSendParam.sendParam.dstEid, message, options
        ...

    }
```

4. The problem is found in this final step. As we can see, if a compose message is detected inside `_buildOFTMsgAndOptions` when `encode` is called, the `msg.sender` will directly be appended as the source chain sender:

```
// BaseTapiocaOmnichainSender.sol

function _buildOFTMsgAndOptions(
        SendParam calldata _sendParam,
        bytes calldata _extraOptions,
        bytes calldata _composeMsg,
        uint256 _amountToCreditLD
    ) internal view returns (bytes memory message, bytes memory opt
        bool hasCompose;

        ...
        (message, hasCompose) = OFTMsgCodec.encode(
            _sendParam.to,
            _toSD(_amountToCreditLD),
            _composeMsg
        );
        ...
        }
    }

// OFTMsgCodec.sol

function encode(
        bytes32 _sendTo,
        uint64 _amountShared,
        bytes memory _composeMsg
    ) internal view returns (bytes memory _msg, bool hasCompose) {
        hasCompose = _composeMsg.length > 0;
        // @dev Remote chains will want to know the composed functi
        _msg = hasCompose
            ? abi.encodePacked(_sendTo, _amountShared, addressToByte
            : abi.encodePacked(_sendTo, _amountShared);
    }
```

As shown, LayerZero's `OFTMsgCodec` helper (used in Tapioca's core) always encodes the `msg.sender` as the source chain sender in compose calls. This means that compose calls triggered from Magnetar will always have Magnetar wrongly set as the cross-chains sender, instead of the user, which completely breaks compose messages functionality.

Recommendation

One possible way to fix this is to pass an arbitrary parameter in `sendPacket` with the desired source chain sender, so that it can be encoded via the `encode` function. However, this should only be allowed if the call comes from Magnetar. Iniside Magnetar, the arbitrary parameter should be set as `msg.sender` (the user interacting with Magnetar). This will make the call properly set the source chain sender, even if it is called from the periphery.

[H-07] - Allowing market's `execute` function to be triggered can be leveraged to exploit user's approvals in several ways

Context: Magnetar.sol#L328

Severity: High

Users will usually approve Magnetar to perform certain operations. Most operations perform direct validation of the caller using the `_checkSender` function. This prevents malicious users from exploiting such approvals or performing unauthorized interactions on behalf of other users.

However, Magnetar's internal `_processMarketOperation` allows the `execute` funciton in BigBang and Singularity to be triggered. As the following code snippet shows, caller validations are only performed if the action to be triggered is `addAsset` or `removeAsset` , but not `execute` :

```
// Magnetar.sol

function _processMarketOperation(
        address _target,
        bytes calldata _actionCalldata,
        uint256 _actionValue,
        bool _allowFailure
    ) private {
        if (!cluster.isWhitelisted(0, _target)) revert Magnetar_Not

        /// @dev owner address should always be first param.
        bytes4 funcSig = bytes4(_actionCalldata[:4]);
        if (funcSig == ISingularity.addAsset.selector || funcSig ==
            /// @dev Owner param check. See Warning above.
            _checkSender(abi.decode(_actionCalldata[4:36], (address
        }
        if (
            funcSig == IMarket.execute.selector || funcSig == ISingu
                || funcSig == ISingularity.removeAsset.selector
        ) {
            _executeCall(_target, _actionCalldata, _actionValue, _a
            return;
        }
        revert Magnetar_ActionNotValid(MagnetarAction.Market, _actio
    }
```

Because no validation is performed when `execute` is called, users can perform arbitrary calls to a market on behalf of other users, exploiting their approvals to Magnetar and effectively stealing all of their assets.

Recommendation

Because the `execute` function acts as an entry point for all functions available in the markets, performing proper input validation should be performed when `execute` is called. This will prevent approvals from being exploited by unauthorized callers.

Medium

[M-01] - Reverting approval instead of setting it will prevent collateral from being added to BigBang

Context: MagnetarMintCommonModule.sol#L289

Severity: Medium

When executing the internal `_depositYBBorrowBB` inside the `MagnetarMintCommonModule` , a call to `revertYieldBoxApproval` is performed instead of actually setting the approval. This will prevent all calls that aim at adding collateral to BigBang to fail, because the approval will always be missing:

```
// MagnetarMintCommonModule.sol

function _depositYBBorrowBB(
        IMintData memory mintData,
        address bigBangAddress,
        IYieldBox yieldBox_,
        address user,
        address marketHelper
    ) internal {

    ...

     if (mintData.collateralDepositData.amount > 0) {
                // _setApprovalForYieldBox(address(bigBang_), yieldB
                _executeDelegateCall(
                    magnetarBaseModuleExternal,
                    abi.encodeWithSelector(
                        MagnetarBaseModuleExternal.revertYieldBoxApp
                    )
                );

       ...

      }

    }
```

Recommendation

Perform an approval to YieldBox, instead of reverting the YieldBox approval.

[M-02] - YieldBox withdrawals hardcode the `from` address to be Magnetar, which prevents users from actually withdrawing their yieldbox shares

Context: MagnetarMintCommonModule.sol#L289

Severity: Medium

When a collateral withdrawal to the current chain is requested in the `AssetModule`, the internal `_withdrawHere` function from the asset module will be called. This function will withdraw YieldBox shares and transfer them to the user. However, as the following snippet shows, Magnetar sets a wrong `from` address when YB's `withdraw` function is called:

```
// MagnetarBaseModule.sol
function _withdrawHere(IYieldBox _yieldBox, uint256 _assetId, bytes3
    _yieldBox.withdraw(_assetId, address(this), OFTMsgCodec.bytes32T
}
```

As shown, instead of setting the user's address as the `from` address from which assets should be withdrawn, `address(this)` is set instead. This makes assets be withdrawn from a wrong address, making it impossible to use Magnetar to withdraw from Yieldbox.

Recommendation

Set the proper user address as the `from` parameter in YieldBox's `withdraw` function

[M-03] - Missing approvals to Pearlmit leads to DoS

Context: Magnetar.sol#L376

Severity: Medium

Some operations in Magnetar interact with Pearlmit to perform token approvals. Pearlmit acts as a PERMIT2 contract, where transfers of tokens require two approvals (assuming permit operations are not used):

1. Approving an asset to be transferred by Pearlmit
2. Approving the target in Pearlmit to transfer such asset

However, Magnetar flows that interact with Pearlmit to approve the targets miss the first step. Let's take, for example, the `lock` flow triggered inside `_processTapLockOperation`:

```
// Magnetar.sol
function _processTapLockOperation(
        address _target,
        bytes calldata _actionCalldata,
        uint256 _actionValue,
        bool _allowFailure
    ) private {

        ...

         if (funcSig == ITapiocaOptionLiquidityProvision.lock.selec
            ...
            address yieldBox = ITapiocaOptionLiquidityProvision(_ta

            ...

            pearlmit.approve(yieldBox, assetId, _target, amount, (b

            IYieldBox(yieldBox).setApprovalForAll(_target, true);

            _executeCall(_target, _actionCalldata, _actionValue, _a
            IYieldBox(yieldBox).setApprovalForAll(_target, true);
            return;
        }

   }
```

The snippet shows how, when locking YieldBox shares via Magnetar, the only approval performed in Pearlmit is approving the `_target` to transfer the YB shares via Peralmit. After that, a regular YB approval is made to `_target`, but this is not the correct approval to be performed because the `TapiocaOptionLiquidityProvision` will try to transfer the YB shares from Mangetar via Pearlmit. Instead, the YB shares should be approved to Pearlmit so that the target can transfer YB shares on behalf of Magnetar using the Pearlmit contract.

Recommendation

In all flows where Pearlmit is used in the target in order to perform token transfers, perform the two approvals mentioned in the issue:

- Trigger a pearlmit approval for the specific asset to be transferred
- Actually approve Pearlmit to transfer the corresponding asset

Low

[L-01] - Setting `allowFailure` to `true` could lead to funds remaining locked forever in some situations

Context: General

Severity: Low

When interacting with Magnetar, users can set the `_allowFailure` flag to true. This prevents the Magnetar execution from failing when low-level calls revert. Although this can be useful to perform several batched operations, users that set `_allowFailure` to true can get their funds locked in Magnetar without possibility of retrieving them. This can happen, for example, when calling the internal `_processTapLockOperation` with the intention to trigger TwTap's `participate` function. As the following snippet shows, tokens will be transferred from `msg.sender` to Magnetar, and then the call to the module will be performed. However, if the call fails and user has set `_allowFailure` to `true`, the funds will remain stuck in Magnetar:

```
// Magnetar.sol
function _processTapLockOperation(
        address _target,
        bytes calldata _actionCalldata,
        uint256 _actionValue,
        bool _allowFailure
    ) private {

    ...

    if (funcSig == ITwTap.participate.selector) {
            (, uint256 amount,) = abi.decode(_actionCalldata[4:], (
            address tapOFT = ITwTap(_target).tapOFT();

            {
                bool isErr = pearlmit.transferFromERC20(msg.sender,
                if (isErr) {
                    revert Magnetar_PearlmitTransferFailed();
                }
            }

            pearlmit.approve(tapOFT, 0, _target, amount.toUint200()
            _executeCall(_target, _actionCalldata, _actionValue, _a
            return;
        }

    ...


   }
```

Recommendation

Although the `_allowFailure` can be a useful feature, calls executed after transferring tokens from user to Magnetar should never be allowed to fail. In those situations, `false` should be hardcoded to prevent funds being stuck in Magnetar.

[L-02] - It is possible to perform arbitrary approvals on behalf of Magnetar by leveraging `setApprovalForAll` and `setApprovalForAsset` calls when processing permit operations

Context: Magnetar.sol#L230

Severity: Low

Magnetar’s `_processPermitOperation` to process permit operations allows approvals to be called for any whitelisted asset whitelisted by the Cluster:

```
// Magnetar.sol

 function _processPermitOperation(address _target, bytes calldata _acti


     ...
     if (
        funcSig == IPermitAll.permitAll.selector || funcSig == IPermitA
         || funcSig == IPermit.permit.selector || funcSig == IPermit.re
         || funcSig == IYieldBox.setApprovalForAll.selector || funcSig
         || funcSig == IPearlmit.permitBatchApprove.selector
        ) {
            // No need to send value on permit
            _executeCall(_target, _actionCalldata, 0, _allowFailure);
            return;
        }

        ...
```

The problem with allowing this kind of approval to be performed is that these approvals will be performed on behalf of Magnetar instead of the user. Although Magnetar is not designed to hold tokens, a malicious user could still force Magnetar to approve any whitelisted token in Tapioca to an arbitrary address, and leverage it to steal tokens if any token ends up being stuck in Magnetar.

Recommendation

Remove support for `setApprovalForAll` and `setApprovalForAsset` in `_processPermitOperation`, as this call won’t actually be used for users.

```
// Magnetar.sol

function _processPermitOperation(address _target, bytes calldata _actio


     ...
     if (
        funcSig == IPermitAll.permitAll.selector || funcSig == IPermitA
         || funcSig == IPermit.permit.selector || funcSig == IPermit.re
-        || funcSig == IYieldBox.setApprovalForAll.selector || funcSig
         || funcSig == IPearlmit.permitBatchApprove.selector
        ) {
```

[L-03] - `MagnetarModuleExtender` actions can't be executed due to all action ID's being validated

Context: Magnetar.sol#L161-L174

Severity: Low

When `burst` is triggered in Magnetar, the `_action.id` supplied by the user will be compared with all the values stored in the `MagnetarAction` enum. If no action has been found, the `magnetarModuleExtender` will be called if set:

```
// Magnetar.sol

function burst(MagnetarCall[] calldata calls) external payable {

    ...

    if (_action.id == MagnetarAction.YieldBoxModule) {
        _executeModule(MagnetarModule.YieldBoxModule, _action.call);
        continue; // skip the rest of the loop
    }

// If no valid action was found, use the Magnetar module extender. Only
    if (
        address(magnetarModuleExtender) != address(0)
          && magnetarModuleExtender.isValidActionId(uint8(_action.id))
      ) {
            bytes memory callData = abi.encodeWithSelector(IMagnetarMod
            (bool success, bytes memory returnData) = address(magnetarM
            if (!success) {
              _getRevertMsg(returnData);
            }
    } else {
    // If no valid action was found, revert
        revert Magnetar_ActionNotValid(_action.id, _action.call);
    }

    ...
```

The problem is that the `_action.id` value supplied by the user is a `MagnetarAction` enum itself. This means that it is impossible to supply an id higher than the values stored in the enum, so the situation where an ID has not been found is simply impossible to occur, given that supplying and invalid ID will cause a revert for the user.

Recommendation

If the intention is to keep the `magnetarModuleExtender` contract for future actions, allow the user to supply a `uint8` as parameter for the action ID, instead of a `MagnetarAction` enum. Otherwise, simply remove the `magnetarModuleExtender` logic.

[L-04] - `MagnetarAssetModule` computes `repayPart` with an outdated rate

Context: MagnetarAssetModule.sol#L103

Severity: Low

Users can trigger a repay action in the `MagnetarAssetModule` by calling its `depositRepayAndRemoveCollateralFromMarket` function and specifying a `repayAmount` greater than zero in the `data` passed as parameter. On repaying, the amount specified will be converted to part by calling the `Helper` 's `getBorrowPartForAmount` so that it can be repaid:

```
// MagnetarAssetModule.sol

function depositRepayAndRemoveCollateralFromMarket(DepositRepayAndRemov
        public
        payable
    {

    ...

     if (data.repayAmount > 0) {
             uint256 repayPart = helper.getBorrowPartForAmount(data.mar

              ...

        }
}
```

The problem is that this conversion won't actually use the latest rate data, given that the `Helper` contract does not trigger the market's `accrue` function in `getBorrowPartForAmount` :

```
// MagnetarHelper.sol

function getBorrowPartForAmount(IMarket market, uint256 amount) externa
        Rebase memory _totalBorrowed;
        (uint128 totalBorrowElastic, uint128 totalBorrowBase) = market.
        _totalBorrowed = Rebase(totalBorrowElastic, totalBorrowBase);

        return _totalBorrowed.toBase(amount, false);
    }
```

The call to the market's `totalBorrow` function will return outdated data given that interest has not been accrued. This will lead the user to repay a smaller amount that he should.

Recommendation

Trigger the market's accrual function prior to calling `getBorrowPartForAmount` so that the market's `totalBorrow` contains the most updated accrued interest.

Informational

[INFO-01] - Unnecessary type casts

Context: General

Severity: Informational

Some functions perform type casts that are unnecessary. For example, the `market` and `marketHelper` parameters in `MagnetarAsssetModule`'s `depositRepayAndRemoveCollateralFromMarket` function are casted to `address`. However, this is unnecessary, given that these are already of type `address`.

```
// MagnetarAsssetModule.sol
function depositRepayAndRemoveCollateralFromMarket(DepositRepayAndRemov
        public
        payable
    {
        // Check sender
        _checkSender(data.user);

        // Check target
        if (!cluster.isWhitelisted(0, address(data.market))) {
            revert Magnetar_TargetNotWhitelisted(address(data.market));
        }
        if (!cluster.isWhitelisted(0, address(data.marketHelper))) {
            revert Magnetar_TargetNotWhitelisted(address(data.marketHel
        }

        ...
```

Recommendation

Remove unnecessary casts where performed.

[INFO-02] - Remaining TODO's in the code

Context: General

Severity: Informational

Some contracts still contain TODO tags. For example, the `_withdrawToChain` function in `MagnetarBaseModule` contains the following TODO:

```
// MagnetarBaseModule.sol

function _withdrawToChain(MagnetarWithdrawData memory data) internal {

    ...
    // TODO: decide about try-catch here
        if (data.unwrap) {
            _lzCustomWithdraw(
                asset,
                data.lzSendParams,
                data.sendGas,
                data.sendVal,
                data.composeGas,
                data.composeVal,
                data.composeMsgType
            );

     ...

  }
```

Recommendation

Remove the remaining TODO’s in the codebase.

Gas

[G-01] - Call can be directly performed without checking the value

Context: Magnetar.sol#L457-L472

Severity: Gas

Inside Magnetar’s `_executeCall` implementation, a condition is added to decide whether the low-level call should be performed passing value or not, depending in the `_actionValue` supplied by the user:

```
// Magnetar.sol

function _executeCall(address _target, bytes calldata _actionCalldata,
        private
        returns (bytes memory returnData)
    {
        bool success;

        if (_actionValue > 0) {
            (success, returnData) = _target.call{value: _actionValue}(_
        } else {
            (success, returnData) = _target.call(_actionCalldata);
        }

        if (!success && !_allowFailure) {
            _getRevertMsg(returnData);
        }
    }
```

However, this check is unnecessary, given that even if the value is 0 the call will still be properly executed.

Recommendation

Remove the condition and directly execute the low-level call.

[G-02] - Function selector checks can be optimized

Context: General

Severity: Gas

When calling an internal function in Magnetar, the calldata will first be validated to see if some additional actions need to be performed. After that, the calldata will be checked again to actually execute the call. This adds an unnecessary use of gas, that could be simplified.

Recommendation

Simplify the function signature checks so that the multiple if conditions are reduced. As an example, the `_processPermitOperation` could be optimized in the following way:

```
// Magnetar.sol

function _processPermitOperation(address _target, bytes calldata _actio
        if (!cluster.isWhitelisted(0, _target)) revert Magnetar_NotAuth

        /// @dev owner address should always be first param.
        // permit(address owner...)
        // revoke(address owner...)
        // permitAll(address from,..)
        // revokeAll(address from,..)
        // permit(address from,...)
        // setApprovalForAll(address from,...)
        // setApprovalForAsset(address from,...)
        bytes4 funcSig = bytes4(_actionCalldata[:4]);
+        bool selectorValidated;
        if (
            funcSig == IPermitAll.permitAll.selector || funcSig == IPerm
                || funcSig == IPermit.permit.selector || funcSig == IPe
        ) {
            /// @dev Owner param check. See Warning above.
            _checkSender(abi.decode(_actionCalldata[4:36], (address)));
+            selectorValidated = true;
        }

        // IPearlmit.permitBatchApprove(IPearlmit.PermitBatchTransferFr
        if (funcSig == IPearlmit.permitBatchApprove.selector) {
            IPearlmit.PermitBatchTransferFrom memory batch =
                abi.decode(_actionCalldata[4:], (IPearlmit.PermitBatchT

            /// @dev Owner param check. See Warning above.
            _checkSender(batch.owner);
+            selectorValidated = true
        }

        /// @dev no need to check the owner for the rest; it's using `m
        if (
-            funcSig == IPermitAll.permitAll.selector || funcSig == IPe
-                || funcSig == IPermit.permit.selector || funcSig == IP
+               selectorValidated
                || funcSig == IYieldBox.setApprovalForAll.selector || f
-                || funcSig == IPearlmit.permitBatchApprove.selector
        ) {
            // No need to send value on permit
            _executeCall(_target, _actionCalldata, 0, _allowFailure);
            return;
        }

        revert Magnetar_ActionNotValid(MagnetarAction.Permit, _actionCa
    }
```

[G-03] - Unnecessary approvals can be removed due to using Pearlmit

Context: General

Severity: Gas

Some parts of the code perform extra approvals that are not actually required. For example, `_processTapLockOperation` performs multiple extra approvals to `_target` that are not actually required for the function to work.

```
// Magnetar.sol

function _processTapLockOperation(
        address _target,
        bytes calldata _actionCalldata,
        uint256 _actionValue,
        bool _allowFailure
    ) private {
                ...
            pearlmit.approve(yieldBox, assetId, _target, amount, (block

            IYieldBox(yieldBox).setApprovalForAll(_target, true);
            _executeCall(_target, _actionCalldata, _actionValue, _allow
            IYieldBox(yieldBox).setApprovalForAll(_target, true);
            return;
        }
```

Recommendation

Remove the extra approvals performed in the parts of the code where they are not necessary, leveraging Pearlmit where possible so that the approvals required can actually be decreased.

[G-04] - Unnecessary validation inside `burst`

Context: Magnetar.sol#L88

Severity: Gas

When `burst` is called inside Magnetar, each iteration will verify that `_action.call.length > 0` in case that `_action.allowFailure` has been set to `false` by the user:

```
// Magnetar.sol
function burst(MagnetarCall[] calldata calls) external payable {
        uint256 valAccumulator;

        uint256 length = calls.length;

        for (uint256 i; i < length; i++) {
            MagnetarCall calldata _action = calls[i];
            if (!_action.allowFailure) {
                require(
                    _action.call.length > 0,
                    string.concat("Magnetar: Missing call for action wi
                );
            }
            valAccumulator += _action.value;

            ...
}
```

This adds an unnecessary extra gas consumption, given that if the function is not supposed to fail, not having calldata will already make the function fail, which ends up being the same scenario as if the `require` was performed.

Recommendation

Remove the call length check validation inside `burst`.

[G-05] - Deploying a new instance of `TapiocaOmnichainEngineHelper` is too gas-intensive

Context: Magnetar.sol#L88

Severity: Gas

Inside `MagnetarBaseModule`, the `_lzCustomWithdraw` will deploy a `TapiocaOmnichainEngineHelper` contract twice in order to obtain the required lz call data:

```
// MagnetarBaseModule.sol

function _lzCustomWithdraw(
        address _asset,
        LZSendParam memory _lzSendParam,
        uint128 _lzSendGas,
        uint128 _lzSendVal,
        uint128 _lzComposeGas,
        uint128 _lzComposeVal,
        uint16 _lzComposeMsgType
    ) private {
        PrepareLzCallReturn memory prepareLzCallReturn = _prepareLzSend

        TapiocaOmnichainEngineHelper _toeHelper = new TapiocaOmnichainE
        PrepareLzCallReturn memory prepareLzCallReturn2 = _toeHelper.pr
            ITapiocaOmnichainEngine(_asset),
            PrepareLzCallData({
                dstEid: _lzSendParam.sendParam.dstEid,
                recipient: _lzSendParam.sendParam.to,
                amountToSendLD: 0,
                minAmountToCreditLD: 0,
                msgType: _lzComposeMsgType,
                composeMsgData: ComposeMsgData({
                    index: 0,
                    gas: _lzComposeGas,
                    value: prepareLzCallReturn.msgFee.nativeFee.toUint1
                    prevData: bytes(""),
                    prevOptionsData: bytes("")
                }),
                lzReceiveGas: _lzSendGas + _lzComposeGas,
                lzReceiveValue: _lzComposeVal,
                refundAddress: _lzSendParam.refundAddress
            })
        );

        if (msg.value < prepareLzCallReturn2.msgFee.nativeFee) {
            revert Magnetar_GasMismatch(prepareLzCallReturn2.msgFee.nat
        }

        IOftSender(_asset).sendPacket{value: prepareLzCallReturn2.msgFe
            prepareLzCallReturn2.lzSendParam, prepareLzCallReturn2.comp
        );
    }

    function _prepareLzSend(address _asset, LZSendParam memory _lzSendP
        private
        returns (PrepareLzCallReturn memory prepareLzCallReturn)
    {
        TapiocaOmnichainEngineHelper _toeHelper = new TapiocaOmnichainE
        prepareLzCallReturn = _toeHelper.prepareLzCall(
            ITapiocaOmnichainEngine(_asset),
            PrepareLzCallData({
                dstEid: _lzSendParam.sendParam.dstEid,
                recipient: _lzSendParam.sendParam.to,
                amountToSendLD: _lzSendParam.sendParam.amountLD,
                minAmountToCreditLD: _lzSendParam.sendParam.minAmountLD
                msgType: 1, // SEND
                composeMsgData: ComposeMsgData({
                    index: 0,
                    gas: 0,
                    value: 0,
                    data: bytes(""),
                    prevData: bytes(""),
                    prevOptionsData: bytes("")
                }),
                lzReceiveGas: _lzSendGas,
                lzReceiveValue: _lzSendVal,
                refundAddress: _lzSendParam.refundAddress
            })
        );
    }
```

As the snippet shows, the `_prepareLzSend` will deploy a new instance of `TapiocaOmnichainEngineHelper`. Then, inside `_lzCustomWithdraw`, another `TapiocaOmnichainEngineHelper` will be deployed to compute the second LZ send data. This is extremely gas intensive, and can be drastically reduced by deploying only one instance of the helper, and reusing it each time the LZ send data needs to be created.

Recommendation

Deploy a single instance of `TapiocaOmnichainEngineHelper` and set it as another Tapioca contract that can be queried whenever the LZ send data must be created.