Tapioca DAO - Bar Security Review

Conducted by: 0xadrii

May 15th, 2024 - May 29th, 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 14 days in total, Tapioca DAO engaged with 0xadrii to review Tapioca Bar. In this period of time, a total of 13 issues were found.

3.1 Overview

Project	Tapioca Bar
Repository	https://github.com/Tapioca-DAO/Tapioca-bar
Commit hash	71558e5a830a194c72ef4a9ef10a0f0997a3851e
Audit timeline	May 15th, 2024 - May 29th, 2024

3.2 Issues found

Severity	# of issues
High	4
Medium	4
Low	3
Informational	2
Total issues	13

4 Findings summary

ID	Description	Status
H-01	Lack of receive function in MarketLiquidatorReceiver will make unwrapping native tOFTs always fail	Fixed in PR #433
H-02	Arbitrary external call to user-supplied swapper can lead to approvals being exploited	Fixed in PR #319
H-03	Wrongly changing `amountIn` for `amountOut` will make all transactions for non-tOFT tokens fail when leveraging	Fixed in BaseLeverageExecutor.sol#L152
H-04	Tapioca omnichain extender flow allows performing delegatecalls to arbitrary addresses	Fixed in PR #304
M-01	Current logic for minting the open interest debt is flawed and could lead to issues with USDO's supply	Fixed in PR #448
M-02	Swapped amount should be `unwrapped` instead of `collateralAmount`	Fixed in PR #442
M-03	Sending all `msg.value` in some compose messages prevents next appended messages from being executed	Fixed in PR #314, PR #432 and PR #216
M-04	Cross-chain message pausing can be bypassed by triggering `OFT`'s default `send` operation	Fixed in PR #215
L-01	`safeTransfer` should be used instead of `transfer`	Fixed in PR #320
L-02	Difference between source chain requested amount and debited amount should be sent to owner instead of source chain sender	Fixed in PR #305
	Attackers can DoS the singularityMarkets	

L-03	and bigBangMarkets view functions by maliciously filling the clonesOf array	Fixed in PR #431
I-01	Unnecessary paused() check	Fixed in PR #460
I-02	AssetTotsDaiLeverageExecutor and SimpleLeverageExecutor are not pausable, although AssetToSGLPLeverageExecutor is	Fixed in PR #459

5 Findings

High Risk

H-01. Lack of receive function in MarketLiquidatorReceiver will make unwrapping native tOFTs always fail

Summary

Because the MarketLiquidatorReceiver lacks a receive function, unwrapping tOFT's that have native asset as underlying will always fail, given that ETH transfers can't be received by MarketLiquidatorReceiver.

Vulnerability detail

When triggering the `onCollateralReceiver` function, the `MarketLiquidatorReceiver` will unwrap the received `tOFT` so that it can be swapped:

```
// MarketLiquidatorReceiver.sol
function onCollateralReceiver(
        address initiator,
        address tokenIn,
        address tokenOut,
        uint256 collateralAmount,
        bytes calldata data
    ) external returns (bool) {
       ...


        // unwrap TOFT
        uint256 unwrapped = ITOFT(tokenIn).unwrap(address(this),
collateralAmount);
        if (unwrapped < swapData.minAmountOut) revert NotEnough();

        ...
}
```

However, if the `tOFT` corresponds to a native asset, the unwrap won't succeed, given that unwrapping native `tOFT`'s involves transferring ETH:

```
// BaseTOFT.sol

function _unwrap(address _toAddress, uint256 _amount) internal virtual {
        _burn(msg.sender, _amount);
        vault.withdraw(_toAddress, _amount);
    }
```

```
// TOFTVault.sol

function withdraw(address to, uint256 amount) external onlyOwner {
        _withdraw(to, amount);
}


 function _withdraw(address to, uint256 amount) private {
        if (amount > viewSupply()) revert AmountNotRight();
        if (_isNative) {
            (bool success,) = to.call{value: amount}("");
            if (!success) revert Failed();
        } else {
            IERC20(_token).safeTransfer(to, amount);
        }
    }
```

However, the `MarketLiquidatorReceiver` lacks a `receive` function, so it is impossible for it to receive the ETH when unwrapping.

Lines of code

[MarketLiquidatorReceiver.sol#L83]()

Impact

High. Bad debt liquidations won't be able to be performed for native `tOFT` 's.

Recommendation

Add a `receive` function in the `MarketLiquidatorReceiver` contract.

H-02. Arbitrary external call to user-supplied swapper can lead to approvals being exploited

Summary

The swap performed when leveraging is carried out via a user-supplied swapper contract. This address is not validated in `ZeroXSwapper`, leading to attacks that aim at draining a users' approval.

Vulnerability detail

When leveraging via `buyCollateral` in `BBLeverage`, users can supply an arbitrary `from` amount from whom tokens will be extracted.
There's two ways to obtain tokens from the `from address`:

- Via the `supplyAmount` parameter: this will directly transfer `asset` from `from` and transfer it to the `leverageExecutor`
- Via the `borrowAmount` paramter: this will force `from` to increase its borrow position, borrowing `asset`

The attack will be carried out by only providing a `supplyAmount`, but only 1 wei of `borrowAmount` so that `from`'s position does not increase and is unhealthy. Checking `buyCollateral`, the steps will be:

1. Swap `asset` transferred from `from` to `collateral`
2. Deposit `amountOut` (amount obtained after swapping) into YieldBox. `collateralShare` is the amount of collateral currently held
3. Ensure `from` has approved `msg.sender` to **borrow** `collateralShare` (note `collateralShare` is obtained from the swapped amount)

```
// BBLeverage.sol

function buyCollateral(address from, uint256 borrowAmount, uint256
supplyAmount, bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageBuy)
        solvent(from)
        notSelf(from)
        returns (uint256 amountOut)
    {
       ...

        {
            uint256 supplyShare = yieldBox.toShare(assetId,
calldata_.supplyAmount, true);
            if (supplyShare > 0) {
                (memoryData.supplyShareToAmount,) =
                    yieldBox.withdraw(assetId, calldata_.from,
address(leverageExecutor), 0, supplyShare);
            }
        }

       ...
        {
            amountOut = leverageExecutor.getCollateral(
                address(asset),
                address(collateral),
                memoryData.supplyShareToAmount +
memoryData.borrowShareToAmount,
                calldata_.data
            );
        }

        uint256 collateralShare = yieldBox.toShare(collateralId, amountOut,
false);

        address(collateral).safeApprove(address(yieldBox),
type(uint256).max);
        yieldBox.depositAsset(collateralId, address(this), calldata_.from,
0, collateralShare);
        address(collateral).safeApprove(address(yieldBox), 0);

        if (collateralShare == 0) revert CollateralShareNotValid();
        _allowedBorrow(calldata_.from, collateralShare);
        ...
    }
```

Let's imagine a scenario where `from` has approved `msg.sender` to transfer 5 tokens (a minimum of 1 wei of borrow approval is required for the attack to be performed). A malicious `msg.sender` can leverage the Leverage flow to steal all the `from`'s USDO. The attack consists of:

1. Calling `buyCollateral` setting the victim as `from`, with 1 wei as `borrowAmount` and `from`'s USDO balance as `supplyAmount`
2. Swapping the USDO for collateral. Because the `swapTarget` is not validated in `ZeroXSwapper`, a user can pass an arbitrary amount that keeps most of `from`'s USDO balance, and only returns 1 or 2 wei of collateral so that the whole transaction succeeds.

```
// ZeroXSwapper.sol
 function swap(SZeroXSwapData calldata swapData, uint256 amountIn, uint256
minAmountOut)
        public
        payable
        returns (uint256 amountOut)
    {
        ...
        // Transfer tokens to this contract
        swapData.sellToken.safeTransferFrom(msg.sender, address(this),
amountIn);

        uint256 amountInBefore =
swapData.sellToken.balanceOf(address(this));
        // Approve the 0x proxy to spend the sell token, and call the swap
function
        swapData.sellToken.safeApprove(swapData.swapTarget, amountIn);
        (bool success,) = swapData.swapTarget.call(swapData.swapCallData);
        if (!success) revert SwapFailed();
        swapData.sellToken.safeApprove(swapData.swapTarget, 0);
       ...

}
```

3. Making `amountOut` return 1 or 2 wei, the `_allowedBorrow` check will pass, even if we have transferred `from`'s full USDO balance.

Lines of code

[ZeroXSwapper.sol#L71-L76]

Impact

High. Approvals can be exploited via the leverage module to steal more assets than the approved ones.

Recommendation

Validate the `swapTarget` address in `ZeroXSwapper`:

```
// ZeroXSwapper.sol

function swap(SZeroXSwapData calldata swapData, uint256 amountIn, uint256
minAmountOut)
        public
        payable
        returns (uint256 amountOut)
    {
        if (!cluster.isWhitelisted(0, msg.sender)) revert
SenderNotValid(msg.sender);
+       if (swapData.swapTarget != zeroXProxy) revert
SwapperNotValid(swapData.swapTarget);

        // Transfer tokens to this contract
        swapData.sellToken.safeTransferFrom(msg.sender, address(this),
amountIn);

        uint256 amountInBefore =
swapData.sellToken.balanceOf(address(this));
        ...
}
```

Another option is to directly use the `zeroXProxy` and not allow users to supply a custom swapper address.

H-03. Wrongly changing `amountIn` for `amountOut` will make all transactions for non-tOFT tokens fail when leveraging

Summary

Changing `amountIn` for `amountOut` in `BaseLeverageExecutor` will lead to DoS for non-tOFT tokens.

Vulnerability detail

When performing a swap in `BaseLeverageExecutor`, the `amountOut` variable is approved to be swapped via 0x:

```
// BaseLeverageExecutor.sol

function _swapAndTransferToSender(
        bool sendBack,
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        bytes memory data
    ) internal returns (uint256 amountOut) {
        ...

        // If the tokenIn is a tOFT, unwrap it. Handles ETH and ERC20.
        if (swapData.toftInfo.isTokenInToft) {
            (tokenIn, amountOut) = _handleToftUnwrap(tokenIn, amountIn);
        }

        // Approve the swapper to spend the tokenIn, and perform the swap.
        tokenIn.safeApprove(address(swapper), amountOut);
        IZeroXSwapper.SZeroXSwapData memory swapperData =
            abi.decode(swapData.swapperData,
(IZeroXSwapper.SZeroXSwapData));
        ...
    }
```

As we can see, `amountOut` is only updated if the token to swap is a `tOFT` (i.e `swapData.toftInfo.isTokenInToft` is `true`). However, for non-tOFT tokens, `amountOut` will be 0, making the swap always fail due to the lack of approval.

Lines of code

[BaseLeverageExecutor.sol#L155]()

Impact

High. Leverage won't work for most of the tokens used in the protocol.

Recommendation

Keep the `amountIn` as the parameter to approve and swap:

```
// BaseLeverageExecutor.sol

function _swapAndTransferToSender(
        bool sendBack,
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        bytes memory data
    ) internal returns (uint256 amountOut) {
        ...

        // If the tokenIn is a tOFT, unwrap it. Handles ETH and ERC20.
        if (swapData.toftInfo.isTokenInToft) {
-           (tokenIn, amountOut) = _handleToftUnwrap(tokenIn, amountIn);
+           (tokenIn, amountIn) = _handleToftUnwrap(tokenIn, amountIn);
        }

        // Approve the swapper to spend the tokenIn, and perform the swap.
-       tokenIn.safeApprove(address(swapper), amountOut);
+       tokenIn.safeApprove(address(swapper), amountIn);
        IZeroXSwapper.SZeroXSwapData memory swapperData =
            abi.decode(swapData.swapperData,
(IZeroXSwapper.SZeroXSwapData));
        ...
    }
```

H-04. Tapioca omnichain extender flow allows performing delegatecalls to arbitrary addresses

Summary

Missing validation for user-supplied data allows performing external calls to arbitrary addresses.

Vulnerability details

Some compose message types will allow users to delegate call `TapiocaOmnichainExtExec` and execute certain functions. This is done inside `TapiocaOmnichainReceiver`'s `_extExec` internal function:

```
// File: TapiocaOmnichainReceiver.sol

function _extExec(uint16 _msgType, address _srcChainSender, bytes memory
_data) internal returns (bool) {
        string memory signature = "";
        address sender = address(0);
        if (_msgType == MSG_APPROVALS) {
            // toeExtExec.erc20PermitApproval(_data);
            signature = "erc20PermitApproval(bytes)";
        } else if (_msgType == MSG_NFT_APPROVALS) {
            // toeExtExec.erc721PermitApproval(_data);
            signature = "erc721PermitApproval(bytes)";
        } else if (_msgType == MSG_PEARLMIT_APPROVAL) {
            // toeExtExec.pearlmitApproval(_srcChainSender,_data);
            signature = "pearlmitApproval(address,bytes)";
            sender = _srcChainSender;
        } else if (_msgType == MSG_YB_APPROVE_ALL) {
            // toeExtExec.yieldBoxPermitAll(_data);
            signature = "yieldBoxPermitAll(bytes)";
        } else if (_msgType == MSG_YB_APPROVE_ASSET) {
            // toeExtExec.yieldBoxPermitAsset(_data);
            signature = "yieldBoxPermitAsset(bytes)";
        } else if (_msgType == MSG_MARKET_PERMIT) {
            // toeExtExec.marketPermit(_data);
            signature = "marketPermit(bytes)";
        } else {
            return false;
        }

        bool success;
        if (sender == address(0)) {
            (success,) =
address(toeExtExec).delegatecall(abi.encodeWithSignature(signature, _data));
        } else {
            (success,) =
address(toeExtExec).delegatecall(abi.encodeWithSignature(signature, sender,
_data));
        }
        if (!success) revert ExtExecFailed(signature);
        return true;
    }
```

As we can see, the arbitrary `_data` received by the compose message is directly delegatecalled to `toeExtExec`, without any validation. This is dangerous, given that the lack of checks allows some operations in the `TapiocaOmnichainExtExec` to interact with malicious addresses. Some examples include:

- the `erc20PermitApproval` and `erc721PermitApproval` calls, which directly interact with `approvals[i].token`:

```
// File: TapiocaOmnichainExtExec.sol

function erc20PermitApproval(bytes memory _data) public {
        ERC20PermitApprovalMsg[] memory approvals =
TapiocaOmnichainEngineCodec.decodeERC20PermitApprovalMsg(_data);

        uint256 approvalsLength = approvals.length;
        for (uint256 i = 0; i < approvalsLength;) {
            try IERC20Permit(approvals[i].token).permit(
                approvals[i].owner,
                approvals[i].spender,
                approvals[i].value,
                approvals[i].deadline,
                approvals[i].v,
                approvals[i].r,
                approvals[i].s
            ) {} catch {}
            unchecked {
                ++i;
            }
        }
    }
function erc721PermitApproval(bytes memory _data) public {
        // TODO: encode and decode packed data to save gas
        ERC721PermitApprovalMsg[] memory approvals =
TapiocaOmnichainEngineCodec.decodeERC721PermitApprovalMsg(_data);

        uint256 approvalsLength = approvals.length;
        for (uint256 i = 0; i < approvalsLength;) {
            try ERC721Permit(approvals[i].token).permit(
                approvals[i].spender,
                approvals[i].tokenId,
                approvals[i].deadline,
                approvals[i].v,
                approvals[i].r,
                approvals[i].s
            ) {} catch {}
            unchecked {
                ++i;
            }
        }
    }
```

- the `pearlmitApproval` call, which directly interacts with `peralmit`:

```
// File: TapiocaOmnichainExtExec.sol

function pearlmitApproval(address _srcChainSender, bytes memory _data)
public {
        (address pearlmit, IPearlmit.PermitBatchTransferFrom memory
batchApprovals) =

TapiocaOmnichainEngineCodec.decodePearlmitBatchApprovalMsg(_data);

        batchApprovals.owner = _srcChainSender; // overwrite the owner with
the src chain sender
        // Redundant security measure, just for the sake of it
        try IPearlmit(pearlmit).permitBatchApprove(batchApprovals,
keccak256(abi.encode(_srcChainSender))) {} catch {}
    }
```

Other interactions with `TapiocaOmnichainExtExec` properly validate user's arbitrary data using `_sanitizeTarget`:

```
// File: TapiocaOmnichainExtExec.sol
function yieldBoxPermitAsset(bytes memory _data) public {
        YieldBoxApproveAssetMsg[] memory approvals =

TapiocaOmnichainEngineCodec.decodeArrayOfYieldBoxPermitAssetMsg(_data);

        uint256 approvalsLength = approvals.length;
        for (uint256 i = 0; i < approvalsLength;) {
            _sanitizeTarget(approvals[i].target);
            unchecked {
                ++i;
            }
        }

        _yieldBoxPermitApproveAsset(approvals);
    }
```

Lines of code

TapiocaOmnichainExtExec.sol#L55
TapiocaOmnichainExtExec.sol#L80
TapiocaOmnichainExtExec.sol#L104

Impact

High. Calls to arbitrary external addresses open multiple attack vectors that include stealing assets held in USDO or exploiting approvals.

Recommendation

Sanitize user-supplied targets in `erc20PermitApproval`, `erc721PermitApproval` and `pearlmitApproval`.

Medium Risk

M-01. Current logic for minting the open interest debt is flawed and could lead to issues with USDO's supply

Summary

The current logic to mint USDO as the incurred interest in Big Bang is wrong, and will potentially lead to USDO depegging.

Vulnerability detail

The concept of open interest can be understood as the amount of debt that still has not been reflected in USDO's supply due to the accrual of interest in the market's borrows. Essentially, it is the amount left of USDO supply so that USDO's `totalSupply` matches the total amount of debt created in BigBang.

`Penrose` incorporates a `mintOpenInterestDebt` function so that the open interest debt can be minted, making USDO's supply be balanced according to the total debt. In order to compute the total open interest debt, `mintOpenInterestDebt` will query each of the market's `computeOpenInterestMintable`:

```
// BigBang.sol
function computeOpenInterestMintable() external onlyOwner returns (uint256)
{
        _accrue();
        uint256 toMint = viewOpenInterest();
        if (toMint == 0) {
            debtMinted = totalBorrow.elastic - totalBorrow.base;
        }
        debtMinted += toMint;
        return toMint;
    }

function viewOpenInterest() public view returns (uint256) {
        uint256 debt = totalBorrow.elastic - totalBorrow.base;
        if (debtMinted > debt) {
            return 0;
        }

        return debt - debtMinted;
    }
```

As we can see, `computeOpenInterestMintable` will query `viewOpenInterest`, which returns the difference between the current `totalBorrow`'s `elastic` and `base`, and is substracted from the total `debtMinted` to prevent the same amount of debt from being minted more than once.

Although this implementation is correct, the current design is not and could lead to issues with USDO's supply. The main problem is found when users that perform borrows via Big Bang complete the cycle (borrow --> repay) between two calls to `mintOpenInterstDebt`. Because `computeOpenInterestMintable` takes into account the **current** `totalBorrow` **state**, a borrow and repay performed between calls to `mintOpenInterstDebt` won't be reflected in `totalBorrow`, effectively breaking a core protocol invariant and lead to a reduced amount of USDO supply.

Impact

High. The reported issue could lead to problems with USDO's peg, given that the supply won't correspond to the amount expected by the protocol design.

Lines of code

[Penrose.sol#L317(https://github.com/Tapioca-DAO/Tapioca-bar/blob/GT_86dtkm1wr_Implement-LiquidateBadDebt-market-liquidator-receiver/contracts/Penrose.sol#L317)

Recommendation

One way to fix this issue without adding a new flow and increasing complexity is to have each market track the amount of interest that has been repaid every time a repay is performed, instead of tracking it by substracting elastic and base at a certain point in time. The steps would be:

1. Compute the interest repaid in `BBLLendingCommon`'s `_repay`. `interestRepaid` is a new global variable:

```
// BBLendingCommon.sol
function _repay(address from, address to, bool skim, uint256 part, bool
checkAllowance)
        internal
        returns (uint256 amount)
    {
        if (part > userBorrowPart[to]) {
            part = userBorrowPart[to];
        }
        if (part == 0) revert NothingToRepay();

        if (checkAllowance && msg.sender != from) {
            uint256 partInAmount;
            Rebase memory _totalBorrow = totalBorrow;
            (_totalBorrow, partInAmount) = _totalBorrow.sub(part, true);

            uint256 allowanceShare =
                _computeAllowanceAmountInAsset(to, exchangeRate,
partInAmount, _safeDecimals(asset));
            _allowedLend(from, allowanceShare);
        }

+       uint256 totalInterestBeforeRepayment = totalBorrow.elastic -
totalBorrow.base;

        (totalBorrow, amount) = totalBorrow.sub(part, true);

+       uint256 totalInterestAfterRepayment = totalBorrow.elastic -
totalBorrow.base;

+       if(totalInterestBeforeRepayment > totalInterestAfterRepayment)

+               interestRepaid += totalInterestBeforeRepayment -
totalInterestAfterRepayment;

        userBorrowPart[to] -= part;

       // @dev amount includes the opening & accrued fees
        yieldBox.withdraw(assetId, from, address(this), amount, 0);

        // @dev burn USDO
        IUsdo(address(asset)).burn(address(this), amount);

        emit LogRepay(from, to, amount, part);
    }
```

2. When minting open interest debt, fetch the `interestRepaid` for each market, and mint it following the current flow in `Penrose`'s `mintOpenInterestDebt`. Make sure to clean the `interestRepaid` for each market afterwards:

```
// Penrose.sol

function mintOpenInterestDebt(address twTap) external onlyOwner {
        uint256 sum;
        // compute mintable debt for all BB markets
        // Origins do not produce debt
        uint256 len = allBigBangMarkets.length;
        for (uint256 i; i < len; i++) {
            IBigBang market = IBigBang(allBigBangMarkets[i]);
            if (isMarketRegistered[address(market)]) {
-                sum += market.computeOpenInterestMintable();
+                sum += market.interestRepaid();
+                market.clearInterestRepaid();
            }
        }

        if (sum > 0) {
            //mint against the open interest; supply should be fully minted
now
            IUsdo(address(usdoToken)).mint(address(this), sum);

            //send it to twTap
            uint256 rewardTokenId =
ITwTap(twTap).rewardTokenIndex(address(usdoToken));
            _distributeOnTwTap(sum, rewardTokenId, address(usdoToken),
ITwTap(twTap));
        }
    }
```

```
// Market.sol

+function clearInterestRepaid() external onlyOwner {
+         emit InterestRepaidCleared();
+        interestRepaid = 0;
+    }
```

M-02. Swapped amount should be `unwrapped` instead of `collateralAmount`

Summary

A wrong variable is used to perform the swap in MarketLiquidatorReceiver, which could lead to DoS in some situations.

Vulnerability detail

When the `onCollateralReceived` hook is triggered in `MarketLiquidatorReceiver`, the received `tOFT` is unwrapped and then swapped. Because unwrapping `tOFT`'s could sometimes incur fees, the amount to be swapped should be the amount returned by the `unwrap` `tOFT` action, instead of the `collateralAmount`:

```
// MarketLiquidatorReceiver.sol

 function onCollateralReceiver(
        address initiator,
        address tokenIn,
        address tokenOut,
        uint256 collateralAmount,
        bytes calldata data
    ) external returns (bool) {
        ...

        // unwrap TOFT
        uint256 unwrapped = ITOFT(tokenIn).unwrap(address(this),
collateralAmount);
        if (unwrapped < swapData.minAmountOut) revert NotEnough();

        // get ERC20
        address erc20 = ITOFT(tokenIn).erc20();


            ...

        // swap TOFT.erc20() with `tokenOut`
        IERC20(erc20).safeApprove(assignedSwapper, unwrapped);
        uint256 amountOut =
IZeroXSwapper(assignedSwapper).swap(swapData.data, collateralAmount,
swapData.minAmountOut);

        ...

    }
```

This could lead to issues in the situation where `collateralAmount` is bigger than the `unwrapped` due to the unwrapping fee. Because of the lack of funds in the contract, the swap will always fail, leading to DoS.

Impact

Medium. `MarketLiquidatorReceiver`'s `onCollateralReceiver` will be DoS'ed in some scenarios.

Lines of code

[MarketLiquidatorReceiver.sol#L102]

Recommendation

Swap the `unwrapped` amount, instead of the `collateralAmount`:

```
// MarketLiquidatorReceiver.sol

 function onCollateralReceiver(
        address initiator,
        address tokenIn,
        address tokenOut,
        uint256 collateralAmount,
        bytes calldata data
    ) external returns (bool) {
        ...

        // unwrap TOFT
        uint256 unwrapped = ITOFT(tokenIn).unwrap(address(this),
collateralAmount);
        if (unwrapped < swapData.minAmountOut) revert NotEnough();

        // get ERC20
        address erc20 = ITOFT(tokenIn).erc20();

            ...

        // swap TOFT.erc20() with `tokenOut`
        IERC20(erc20).safeApprove(assignedSwapper, unwrapped);
-        uint256 amountOut =
IZeroXSwapper(assignedSwapper).swap(swapData.data, collateralAmount,
swapData.minAmountOut);
+        uint256 amountOut =
IZeroXSwapper(assignedSwapper).swap(swapData.data, unwrapped,
swapData.minAmountOut);

        ...

    }
```

M-03. Sending all `msg.value` in some compose messages prevents next appended messages from being executed

Sumary

Sending all `msg.value` when triggering some compose messages will prevent next messages from being properly executed if they require value to be sent.

Vulnerability detail

Some compose messages require value to be sent in their transactions. For example, the `_repay` function in `UsdoMarkeReceiverModule` triggered for compose messages of type `MSG_YB_SEND_SGL_LEND_OR_REPAY` will interact with Magnetar and send all the `msg.value` in the transaction to it:

```
// File: UsdoMarkeReceiverModule.sol

function _repay(MarketLendOrRepayMsg memory msg_, address srcChainSender)
private {
        ...
        MagnetarCall[] memory magnetarCall = new MagnetarCall[](1);
        magnetarCall[0] = MagnetarCall({
            id: uint8(MagnetarAction.AssetModule),
            target: msg_.lendParams.magnetar, //ignored in modules call
            value: msg.value,
            call: call
        });
        IMagnetar(payable(msg_.lendParams.magnetar)).burst{value: msg.value}
(magnetarCall);
    }
```

Sending all the transaction's `msg.value` instead of setting a specific amount will lead to issues when multiple compose messages are sent. As seen in the following code snippet, `TapiocaOmnichainReceiver` allows additional compose messages (`nextMsg_`) to be appended and executed when sending a compose message:

```
function _lzCompose(address srcChainSender_, bytes32 _guid, bytes memory
oftComposeMsg_) internal {
        ...

        emit ComposeReceived(msgType_, _guid, tapComposeMsg_);
        if (nextMsg_.length > 0) {
            _lzCompose(srcChainSender_, _guid, nextMsg_);
        }
    }
```

If `nextMsg_` requires an interaction where some ETH value needs to be sent (maybe the second compose message also interacts with magnetar, or needs to send a cross-chain message where some value is required for the fee), the whole transaction will revert and the compose message won't be fulfillable, given that all ETH will be consumed in the previous transaction.

Lines of code

[UsdoMarketReceiverModule.sol#L183]https://github.com/Tapioca-DAO/Tapioca-bar/blob/71558e5a830a194c72ef4a9ef10a0f0997a3851e/contracts/usdo/modules/UsdoMa
UsdoMarketReceiverModule.sol#L219
UsdoMarketReceiverModule.sol#L252
UsdoOptionReceiverModule.sol#L174

Impact

Medium. Dos will take place in some combinations of compose messages due to the lack of value.

Recommendation

In such interactions, instead of transferring the whole `msg.value`, enforce an additional encoded parameter that specifies the required value to be sent in the compose message. This will each chained compose message to only transfer the required value, instead of the whole transaction's value.

M-04. Cross-chain message pausing can be bypassed by triggering `OFT`'s default `send` operation

Summary

USDO's inherited `send` function from `OFTCore` lacks a pausing check, which allows users to trigger cross-chain messages even if USDO is paused.

Vulnerabilty detail

USDO includes a pausing mechanism that prevents some functions from being executed in certain situations. One of the critical functions that needs to be paused is USDO's `sendPacket`, which allows cross-chain packets to be sent (note the `whenNotPaused` modifier):

```
// File: Usdo.sol
function sendPacket(LZSendParam calldata _lzSendParam, bytes calldata
_composeMsg)
        public
        payable
        whenNotPaused
        returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
oftReceipt)
    {
        (msgReceipt, oftReceipt) = abi.decode(
            _executeModule(
                uint8(IUsdo.Module.UsdoSender),
                abi.encodeCall(TapiocaOmnichainSender.sendPacket,
(_lzSendParam, _composeMsg)),
                false
            ),
            (MessagingReceipt, OFTReceipt)
        );
    }
```

However, because USDO inherits from LayerZero's OFT implementation (USDO > BaseUsdo > BaseTapiocaOmnichainEngine > OFT > OFTCore), which has a `send` function that also allows cross-chain messages to be sent, users can bypass the paused status and still transfer assets by leveraging this OFTCore function:

```
// File: OFTCore.sol

function send(
        SendParam calldata _sendParam,
        MessagingFee calldata _fee,
        address _refundAddress
    ) external payable virtual returns (MessagingReceipt memory msgReceipt, 
OFTReceipt memory oftReceipt) {
        // @dev Applies the token transfers regarding this send() operation.
        // - amountSentLD is the amount in local decimals that was ACTUALLY 
sent from the sender.
        // - amountReceivedLD is the amount in local decimals that will be 
credited to the recipient on the remote OFT instance.
        (uint256 amountSentLD, uint256 amountReceivedLD) = _debit(
            _sendParam.amountLD,
            _sendParam.minAmountLD,
            _sendParam.dstEid
        );

        // @dev Builds the options and OFT message to quote in the endpoint.
        (bytes memory message, bytes memory options) = 
_buildMsgAndOptions(_sendParam, amountReceivedLD);

        // @dev Sends the message to the LayerZero endpoint and returns the 
LayerZero msg receipt.
        msgReceipt = _lzSend(_sendParam.dstEid, message, options, _fee, 
_refundAddress);
        // @dev Formulate the OFT receipt.
        oftReceipt = OFTReceipt(amountSentLD, amountReceivedLD);

        emit OFTSent(msgReceipt.guid, _sendParam.dstEid, msg.sender, 
amountSentLD);
    }
```

Lines of code

[Usdo.sol#L45]

Impact

Medium. A core functionality (setting the protocol in a paused state) can be bypassed.

Recommendation

Override OFTCore's `send` function and include the `whenNotPaused` so that cross-chain calls can not be performed when in a paused state.

Low Risk

L-01. `safeTransfer` should be used instead of `transfer`

Summary

`safeTransfer` should be used to transfer the difference back in `ZeroXSwapper`.

Vulnerability detail

Some tokens as USDT will revert if the generic ERC20 interface is used to perform transfers. Although Tapioca only plans to use decentralized network gas tokens and decentralized liquid staking tokens as collateral, it would be good to fix this just in case some of the collateral tokens don't support regular transfers.

Lines of code

ZeroXSwapper.sol#L83

Impact

Low

Recommendation

Use `safeTransfer` to return the difference to the user:

```
// ZeroXSwapper.sol

function swap(SZeroXSwapData calldata swapData, uint256 amountIn, uint256
minAmountOut)
        public
        payable
        returns (uint256 amountOut)
    {
        ...

        // @dev should never be the case otherwise
        if (amountInBefore > amountInAfter) {
            uint256 transferred = amountInBefore - amountInAfter;
            if (transferred < amountIn) {
-               swapData.sellToken.transfer(msg.sender, amountIn -
transferred);
+               swapData.sellToken.safeTransfer(msg.sender, amountIn -
transferred);
            }
        }
```

L-02. Difference between source chain requested amount and debited amount should be sent to owner instead of source chain sender

Summary

Refunded assets when performing remote transfers are sent to the wrong address

Vulnerability detail

When performing an internal remote transfer via compose messages, a situation can arise where the source chain requested amount is bigger than the amount debited in the destination chain. In that situation, the difference will be returned to the source chain sender:

```
// File: `TapiocaOmnichainReceiver.sol`

function _internalRemoteTransferSendPacket(
        address _srcChainSender,
        LZSendParam memory _lzSendParam,
        bytes memory _composeMsg
    ) internal returns (MessagingReceipt memory msgReceipt, OFTReceipt
memory oftReceipt) {
        ...
        if (_lzSendParam.sendParam.amountLD > amountDebitedLD_) {
            // Send the difference back to the user
            _transfer(address(this), _srcChainSender,
_lzSendParam.sendParam.amountLD - amountDebitedLD_);

            // Overwrite the amount to credit with the amount debited
            _lzSendParam.sendParam.amountLD = amountDebitedLD_;
            _lzSendParam.sendParam.minAmountLD =
_removeDust(amountDebitedLD_);
        }
       ...

    }
```

However, returning the difference to `_srcChainSender` is wrong, given that assets were initially transferred from `remoteTransferMsg_.owner` in the call to `_internalTransferWithAllowance`, so the refund should be transferred to `remoteTransferMsg_.owner`:

```
// File: `TapiocaOmnichainReceiver.sol`

function _remoteTransferReceiver(address _srcChainSender, bytes memory
_data) internal virtual {
        RemoteTransferMsg memory remoteTransferMsg_ =
TapiocaOmnichainEngineCodec.decodeRemoteTransferMsg(_data);

        /// @dev xChain owner needs to have approved dst srcChain
`sendPacket()` msg.sender in a previous composedMsg. Or be the same address.
        _internalTransferWithAllowance(
            remoteTransferMsg_.owner, _srcChainSender,
remoteTransferMsg_.lzSendParam.sendParam.amountLD
        );

        // Make the internal transfer, burn the tokens from this contract
and send them to the recipient on the other chain.
        _internalRemoteTransferSendPacket(
            _srcChainSender, remoteTransferMsg_.lzSendParam,
remoteTransferMsg_.composeMsg
        );

        ....
    }

function _internalTransferWithAllowance(address _owner, address
_srcChainSender, uint256 _amount) internal {
        _validateAndSpendAllowance(_owner, _srcChainSender, _amount);
        _transfer(_owner, address(this), _amount);
    }
```

Lines of code

[TapiocaOmnichainReceiver.sol#L260]()

Impact

Low.

Recommendation

Update the refund address to be the `remoteTransferMsg_.owner` parameter specified by the user, instead to the source chain sender.

L-03. Attackers can DoS the singularityMarkets and bigBangMarkets view functions by maliciously filling the clonesOf array

Summary

Attackers can add several markets to `clonesOf` array to DoS some view functions.

Vulnerability detail

Penrose inherits from BoringCrypto's `BoringFactory`, which allows to permissionlessly deploy new contracts given a master contract:

```
// File: BoringFactory.sol

function deploy(
        address masterContract,
        bytes calldata data,
        bool useCreate2
    ) public payable returns (address cloneAddress) {
        require(masterContract != address(0), "BoringFactory: No
masterContract");
        bytes20 targetBytes = bytes20(masterContract); // Takes the first 20
bytes of the masterContract's address

        if (useCreate2) {
            // each masterContract has different code already. So clones are
distinguished by their data only.
            bytes32 salt = keccak256(data);

            // Creates clone, more info here:
https://blog.openzeppelin.com/deep-dive-into-the-minimal-proxy-contract/
            assembly {
                let clone := mload(0x40)
                mstore(clone,
0x3d602d80600a3d3981f3363d3d373d3d3d363d73000000000000000000000000)
                mstore(add(clone, 0x14), targetBytes)
                mstore(add(clone, 0x28),
0x5af43d82803e903d91602b57fd5bf30000000000000000000000000000000000)
                cloneAddress := create2(0, clone, 0x37, salt)
            }
        } else {
            assembly {
                let clone := mload(0x40)
```

```
                mstore(clone,
0x3d602d80600a3d3981f3363d3d373d3d3d363d73000000000000000000000000)
                mstore(add(clone, 0x14), targetBytes)
                mstore(add(clone, 0x28),
0x5af43d82803e903d91602b57fd5bf30000000000000000000000000000000000)
                cloneAddress := create(0, clone, 0x37)
            }
        }
        masterContractOf[cloneAddress] = masterContract;
        clonesOf[masterContract].push(cloneAddress);

        IMasterContract(cloneAddress).init{value: msg.value}(data);

        emit LogDeploy(masterContract, data, cloneAddress);
    }
```

As we can see, there's no restrictions to deploy new contracts, and each new contract will fill the `clonesOf` array.

Knowing this, an attacker can deploy multiple instances of BigBang and Singularity's `masterContract`s, in order to make the array increase to a huge amount. With such array being filled with a lot of data, the `getAllMasterContractClones` will dos, given that `clonseOfCount` will return a huge value.

```
function getAllMasterContractClones(IPenrose.MasterContract[] memory array)
        public
        view
        returns (address[] memory markets)
    {
        uint256 _masterContractLength = array.length;
        uint256 marketsLength = 0;

        unchecked {
            // We first compute the length of the markets array
            for (uint256 i; i < _masterContractLength;) {
                marketsLength += clonesOfCount(array[i].location);

                ++i;
            }
        }

        markets = new address[](marketsLength);

        uint256 marketIndex;
        uint256 clonesOfLength;

        unchecked {
            // We populate the array
            for (uint256 i; i < _masterContractLength;) {
                address mcLocation = array[i].location;
                clonesOfLength = clonesOfCount(mcLocation);

                // Loop through clones of the current MC.
                for (uint256 j = 0; j < clonesOfLength;) {
                    markets[marketIndex] = clonesOf[mcLocation][j];
                    ++marketIndex;
                    ++j;
                }
                ++i;
            }
        }
    }
```

DoS'ing `getAllMasterContractClones` will inevitably DoS `singularityMarkets` and `bigBangMarkets` view functions.

Lines of code

Penrose.sol#L543

Impact

Low. Some view functions could be maliciously DoS'ed.

Recommendation

Override the `deployFor` function so that it is permissioned to only be called by a Penrose admin.

Gas Optimization

No issues found.

Informational

I-01. Unnecessary paused() check

Summary

`USDOFlashLoanHelper` includes a redundant `paused` check that can be removed.

Vulnerability detail

The `flashLoan` function in `USDOFlashLoanHelper` includes a check to ensure that no flash loans can be performed when USDO is paused:

```
// USDOFlashLoanHelper.sol

function flashLoan(IERC3156FlashBorrower receiver, address token, uint256
amount, bytes calldata data)
        external
        override
        returns (bool)
    {
        if (token != address(usdo)) revert NotValid();
        if (usdo.paused()) revert Paused();

       ...
}
```

This check is redundant and can be removed, as USDO's mint function will always fail if USDO is paused:

```
// Usdo.sol
function mint(address _to, uint256 _amount) external whenNotPaused {
        if (!allowedMinter[_getChainId()][msg.sender]) {
            revert Usdo_NotAuthorized();
        }
        _mint(_to, _amount);
}
```

Impact

Informational

Lines of code

[USDOFlashloanHelper.sol#L116]()

Recommendation

Remove the `Paused` check in `flashLoan`.

I-02. AssetTotsDaiLeverageExecutor and SimpleLeverageExecutor are not pausable, although AssetToSGLPLeverageExecutor is

Summary

The `AssetToSGLPLeverageExecutor` contract is pausable. However, `AssetTotsDaiLeverageExecutor` and `SimpleLeverageExecutor` are not.

Vulnerability detail

Both `AssetTotsDaiLeverageExecutor` and `SimpleLeverageExecutor` lack inheriting the `Pausable` contract. This makes them unpausable.

Impact

Informational

Lines of code

AssetTotsDaiLeverageExecutor.sol#L23
SimpleLeverageExecutor.sol#L22

Recommendation

Inherit `Pausable` for both `AssetTotsDaiLeverageExecutor` and `SimpleLeverageExecutor`, like it is done for the `AssetToSGLPLeverageExecutor` leverage executor.