TapiocaDAO Audit Report

Version 1.0

Windhustler

TapiocaDAO Audit Report

Windhustler

May 17, 2024

Introduction

A time-boxed security review of the **TapiocaDAO** protocol was done by **Windhustler**, focusing on the security aspects of the smart contracts.

Disclaimer

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

About Windhustler

Windhustler is an independent smart contract security researcher. Having extensive experience in developing and managing DeFi projects holding millions in TVL, he is putting his best efforts into security research & reviews. Check his previous work here or reach out on X @windhustler.

About TapiocaDAO

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.

Severity classification

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

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

***review commit hash*:**

- **TapiocaZ - 68b41b2606b0c088c86fa3cce8ad4acb622a79aa**
- **Tapioca-bar - e00895fe41d938106c04098bbcbb35d6c23ea300**
- **tap-token - 5a2e9d7a13b412f575d4936080d93dcd0956eb25**

Scope

The following smart contracts were in the scope of the audit:

- `tap-token/contracts/tokens/TapTokenReceiver.sol`
- `Tapioca-bar/contracts/usdo/modules/UsdoMarketReceiverModule.sol`

- Tapioca-bar/contracts/usdo/modules/UsdoOptionReceiverModule.sol
- TapiocaZ/contracts/Balancer.sol
- TapiocaZ/contracts/tOFT/modules/TOFTGenericReceiverModule.sol
- TapiocaZ/contracts/tOFT/modules/TOFTMarketReceiverModule.sol
- TapiocaZ/contracts/tOFT/modules/TOFTOptionsReceiverModule.sol

---

Findings Summary

ID	Title	Severity
H-01	`leverageUpReceiver` depositing assets to yieldBox can be exploited	High
H-02	`twTap` rewards can be stolen in case of pending allowances	High
M-01	amount conversion for `StargatePool` is incorrectly implemented	Medium
M-02	DoS of functionality due to useless approvals in several instances	Medium
M-03	Withdrawing to other chain when exercising options won't work as expected, leading to DoS	Medium
M-04	Withdrawing to other chain when exercising option is non-functional due to absence of `msg.value`	Medium
M-05	`UsdoMarketReceiverModule.lendOrRepayReceiver` repay flow is non-functional	Medium
M-06	Incorrect decoding in `decodeLockTwpTapDstMsg`	Medium
M-07	Adding reward tokens to `twTAP` results in `lzCompose` call permanently reverting	Medium
L-01	PoolIds should be required for all the connectedOFTs	Low
L-02	marketHelper is not whitelisted in removeAssetReceiver flow	Low

Detailed Findings

[H-01] `leverageUpReceiver` depositing assets to yieldBox can be exploited

Context

- TOFTMarketReceiverModule.sol#L179

Description

`leverageUpReceiver` cross-chain flow is intended to be used in two ways:

- Alice initiates a call from `chainA` to `chainB` and she wants to use the leveraging functionality.
- Alice gives Bob approval for a certain amount of TOFT. This gives Bob permission to initiate the cross-chain call and use the functionality on Alice's behalf.

Moreover, the function has two options:

- collateral can be bought with the `TOFT`.
- The asset belonging to the Market can be deposited inside the YieldBox, and used upstream for leveraging operation.

```
## TOFTMarketReceiverModule.sol

if (msg_.supplyAmount > 0) {
    IYieldBox yb = IYieldBox(IMarket(msg_.market)._yieldBox());
    yb.depositAsset(IMarket(msg_.market)._assetId(), msg_.user, msg_.
       user, msg_.supplyAmount, 0);
}
```

The `msg.sender` for the `YieldBox.depositAsset` call is `TOFT`, which means that `msg_.user` needs to give a proper allowance inside the Yieldbox to `TOFT` for this to work.

```
## YieldBox.sol

    function depositAsset(
        uint256 assetId,
        address from,
        address to,
        uint256 amount,
        uint256 share
    )
        public
```

```
>>>         allowed(from, assetId)
        returns (uint256 amountOut, uint256 shareOut)
    {
```

This is an issue because as soon as a user gives this allowance anyone can initiate the cross-chain call, specify the `msg_.user` as our victim and leverage up on his behalf. The damage can be done upstream in the leveraging operation that uses trading.

One would assume this is not possible due to:

```
## TOFTMarketReceiverModule.sol

    function _validateLeverageUpReceiver(LeverageUpActionMsg memory
        msg_, address srcChainSender) private returns (
        LeverageUpActionMsg memory) {

        _checkWhitelistStatus(msg_.market); // we validate the market
            address
        _checkWhitelistStatus(msg_.marketHelper); // we validate the
            marketHelper address

        msg_.borrowAmount = _toLD(msg_.borrowAmount.toUint64());
        if (msg_.supplyAmount > 0) { msg_.supplyAmount = _toLD(msg_.
            supplyAmount.toUint64()); }

>>>         _validateAndSpendAllowance(msg_.user, srcChainSender, msg_.
    borrowAmount);

        return msg_;
    }
```

Where the attacker would be the `srcChainSender` and the victim `msg_.user`. This check can be bypassed by simply passing `msg_.borrowAmount == 0` and then the attacker doesn't have to have any allowance for TOFT from the victim.

Recommendation

Pending allowances from users to the `TOFT` contract are problematic in the current architecture. Rethink the flow and implement the needed changes accordingly.

[H-02] twTap rewards can be stolen in case of pending allowances

Context

- TapTokenReceiver.sol#L161

Description

The precondition for executing `_claimTwpTapRewardsReceiver` as part of the `lzCompose` flow is the `claimTwTapRewardsMsg_.tokenId` owner granting approval for his NFT to the `TapOFT` contract.

```
## TwTAP.sol

function _requireClaimPermission(address _to, uint256 _tokenId)
    internal view {
>>>     if (!_isApprovedOrOwner(_to, _tokenId) && !isERC721Approved(
    _ownerOf(_tokenId), _to, address(this), _tokenId)) {
         revert NotApproved(_tokenId, msg.sender);
     }
}
```

Approval can be granted through the `ERC721` or through Pearlmit with `isERC721Approved(_ownerOf(_tokenId), _to, address(this), _tokenId)`. In case any of the two approvals are given the function will not revert.

This issue was also found in the **Code4rena competitive audit**.

As rewards can be transferred to any arbitrary address, pending allowances can be immediately exploited. It's worth noting here that the usual flow using this functionality is having a composed message, whereby the first message grants approval through permit and the second executes the logic.

The problem is that anyone can execute the permit on the `ERC721` token and send their own cross-chain call to transfer all the `twTAP` rewards.

Recommendation

Remove the `ERC721` approval check and implement broad architectural changes to disallow reusing allowances and submitting permits on the user's behalf.

[M-01] amount conversion for StargatePool is incorrectly implemented

Context

- Balancer.sol#L199

Description

In response to the following issue found in Tapioca's Sherlock contest the `Balancer.sol` was changed to take `StargatePool` conversion rate into account.

The fixes weren't applied properly which introduced further issues.

StargatePools have the concept of `conversionRate` as described in the issue mentioned above. This is the same concept that applies to OFTs from LayerZero.

The logic is the following:

- If the `conversionRate` is different than 1, e.g. if it's `10**12` for DAI StargatePool.
- If the user tries to transfer a value of `1234567890123456789`.
- This exact amount is non-transferrable, the last 12 digits need to be cleaned and only `1234567000000000000` is transferable.

In order to clean the last n digits the amount needs to be first divided and then multiplied by the `convertRate`:

`amount / convertRate * convertRate`.

The way it's done currently:

```
uint256 convertedAmount = _amount;
address stargatePool = stargateFactory.getPool(connectedOFTs[_srcOft][
    _dstChainId].srcPoolId);
uint256 sharedDecimals = IStargatePool(stargatePool).sharedDecimals();
uint256 convertRate = IStargatePool(stargatePool).convertRate();
if (convertRate != 1) {
// ex: for 10e18 and 6 shared decimals => 10e18 / 1e12 * 1e6, 10e12
convertedAmount = (_amount / convertRate) * (10 ** sharedDecimals);
}
```

If you try to rebalance `1234567890123456789` you get the converted amount of:

`1234567890123456789 / 10**12 * 10**6 = 1.2345679e12` which is just a fraction of `1234567e12`.

After the conversion if the token to rebalance is:

- `ETH`, only the converted amount is rebalanced leading to ETH hanging in the `Balancer.sol` contract.
- `ERC20`, the original amount is transferred making the conversion redundant.

Recommendation

```
## Balancer.sol

contract Balancer is Ownable {
     {
          if (msg.sender != owner() && msg.sender != rebalancer) revert
             NotAuthorized();

-         if (connectedOFTs[_srcOft][_dstChainId].rebalanceable <
    _amount) {
-             revert RebalanceAmountNotSet();
-         }
-
          uint256 convertedAmount = _amount;
          address stargatePool = stargateFactory.getPool(connectedOFTs[
             _srcOft][_dstChainId].srcPoolId);
-         uint256 sharedDecimals = IStargatePool(stargatePool).
    sharedDecimals();
          uint256 convertRate = IStargatePool(stargatePool).convertRate
             ();
          if (convertRate != 1) {
-             // ex: for 10e18 and 6 shared decimals => 10e18 / 1e12 * 1
    e6, 10e12
-             convertedAmount = (_amount / convertRate) * (10 **
    sharedDecimals);
+             convertedAmount = (_amount / convertRate) * convertRate;
+         }
+         assert(convertedAmount != 0);
+
+         if (connectedOFTs[_srcOft][_dstChainId].rebalanceable <
    convertedAmount) {
+             revert RebalanceAmountNotSet();
          }

          //extract
-         ITOFT(_srcOft).extractUnderlying(_amount);
+         ITOFT(_srcOft).extractUnderlying(convertedAmount);

          //send
          {
@@ -210,11 +209,11 @@ contract Balancer is Ownable {
                  if (disableEth) revert SwapNotEnabled();
                  _sendNative(_srcOft, convertedAmount, _dstChainId,
                     _slippage);
              } else {
-                 _sendToken(_srcOft, _amount, _dstChainId, _slippage);
+                 _sendToken(_srcOft, convertedAmount, _dstChainId,
    _slippage);
              }

```

```
40 -            connectedOFTs[_srcOft][_dstChainId].rebalanceable -=
       _amount;
41 -            emit Rebalanced(_srcOft, _dstChainId, _slippage, _amount,
       convertedAmount, _isNative);
42 +            connectedOFTs[_srcOft][_dstChainId].rebalanceable -=
       convertedAmount;
43 +            emit Rebalanced(_srcOft, _dstChainId, _slippage,
       convertedAmount, convertedAmount, _isNative);
44          }
45      }
```

[M-02] DoS of functionality due to useless approvals in several instances

Context

- TOFTMarketReceiverModule.sol#L173
- TOFTMarketReceiverModule.sol#L199
- TOFTMarketReceiverModule.sol#L240

Description

`leverageUpReceiver` and other function calls inside the `TOFTMarketReceiverModule` as well as function calls in other modules are called as part of the `lzCompose`. This is invoked by the lzEndpoint to trigger the composed message invocation.

If we take the example of `leverageUpReceiver`, the `msg.sender` during this flow is the lzEndpoint. This means that the `approve(address(msg_.market), type(uint256).max);` sets the allowances for the owner being `lzEndpoint` and the spender being `msg_.market`.

```
1  ## TOFTMarketReceiverModule.sol
2
3  function _marketLeverage(LeverageUpActionMsg memory msg_) private {
4  >>>     approve(address(msg_.market), type(uint256).max);
5      (Module[] memory modules, bytes[] memory calls) = IMarketHelper(
           msg_.marketHelper).buyCollateral(
6          msg_.user, msg_.borrowAmount, msg_.supplyAmount, msg_.
               executorData
7      );
8      if (msg_.supplyAmount > 0) {
9          IYieldBox yb = IYieldBox(IMarket(msg_.market)._yieldBox());
10         yb.depositAsset(IMarket(msg_.market)._assetId(), msg_.user,
               msg_.user, msg_.supplyAmount, 0);
11     }
12     IMarket(msg_.market).execute(modules, calls, true);
```

```
13      approve(address(msg_.market), 0);
14  }
```

As lzEndpoint most certainly is not going to hold any TOFT tokens this allowance is useless. On the other hand, the Market does have to pull some funds from the `msg_.user` so if it is not provided otherwise this function is going to revert.

Other instances

- `marketBorrowReceiver` function:

```
## TOFTMarketReceiverModule.sol

function _marketBorrow(MarketBorrowMsg memory msg_) private {
>>>     approve(address(msg_.borrowParams.magnetar), msg_.borrowParams.
    amount);

    bytes memory call = abi.encodeWithSelector(
        MagnetarCollateralModule.
           depositAddCollateralAndBorrowFromMarket.selector,
        DepositAddCollateralAndBorrowFromMarketData(
            msg_.borrowParams.market,
            msg_.borrowParams.marketHelper,
            msg_.user,
            msg_.borrowParams.amount,
            msg_.borrowParams.borrowAmount,
            msg_.borrowParams.deposit,
            msg_.withdrawParams
        )
    );
    MagnetarCall[] memory magnetarCall = new MagnetarCall[](1);
    magnetarCall[0] = MagnetarCall({
        id: uint8(MagnetarAction.CollateralModule),
        target: msg_.borrowParams.market,
        value: msg.value,
        call: call
    });
    IMagnetar(payable(msg_.borrowParams.magnetar)).burst{value: msg.
       value}(magnetarCall);
}
```

- `marketRemoveCollateralReceiver` function:

```
function _marketRemoveCollateral(MarketRemoveCollateralMsg memory msg_)
    private {
    address ybAddress = IMarket(msg_.removeParams.market)._yieldBox();
    uint256 assetId = IMarket(msg_.removeParams.market)._collateralId()
        ;
```

```
4
5       uint256 share = IYieldBox(ybAddress).toShare(assetId, msg_.
            removeParams.amount, false);
6   >>>     approve(msg_.removeParams.market, share);
7
8       (Module[] memory modules, bytes[] memory calls) = IMarketHelper(
            msg_.removeParams.marketHelper)
9           .removeCollateral(msg_.user, msg_.withdrawParams.withdraw ?
                msg_.removeParams.magnetar : msg_.user, share);
10      IMarket(msg_.removeParams.market).execute(modules, calls, true);
11  }
```

Recommendation

Remove the redundant allowances and implement a system where the user can give the needed approvals.

[M-03] Withdrawing to other chain when exercising options won't work as expected, leading to DoS

Context

- TOFTOptionsReceiverModule.sol#L235

Description

During the Sherlock competitive audit the following issue has been found:

- https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/125
- https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/92

In summary, during `exerciseOption` flow, `TapOFT` is obtained but the logic tries to transfer the `TOFT`. As the `msg.sender` is the `lzEndpoint` this will always fail.

As indicated by this ***comment*** the team has fixed this issue inside the `UsdoOptionReceiverModule` but the fix is not present in the context of `TOFTOptionsReceiverModule.sol`.

Recommendation

Implement the recommended solution of sending `TapOFT` instead of `TOFT`.

[M-04] Withdrawing to other chain when exercising options won't work as expected, leading to DoS

Context

- UsdoOptionReceiverModule.sol#L164

Description

In case when the user using the `exerciseOption` flow wants to transfer the obtained `TapOFT` to another chain he needs to pay for the `lzSend`.

```
## UsdoOptionReceiverModule.sol

msg_.lzSendParams.sendParam = _send;
>> IOftSender(tapOft).sendPacket(msg_.lzSendParams, "");

## TapToken.sol

function sendPacket(LZSendParam calldata _lzSendParam, bytes calldata
    _composeMsg)
  public
  payable
  returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
     oftReceipt)
{
  (msgReceipt, oftReceipt) = abi.decode(
      _executeModule(
          uint8(ITapToken.Module.TapTokenSender),
          abi.encodeCall(TapiocaOmnichainSender.sendPacket, (
              _lzSendParam, _composeMsg)),
          false
      ),
      (MessagingReceipt, OFTReceipt)
  );
}

## TapiocaOmnichainSender.sol

function sendPacket(LZSendParam calldata _lzSendParam, bytes calldata
    _composeMsg)
  external
  payable
  returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
     oftReceipt)
{
  // @dev Applies the token transfers regarding this send() operation.
```

```
31    // - amountDebitedLD is the amount in local decimals that was
          ACTUALLY debited from the sender.
32    // - amountToCreditLD is the amount in local decimals that will be
          credited to the recipient on the remote OFT instance.
33    (uint256 amountDebitedLD, uint256 amountToCreditLD) = _debit(
34        msg.sender,
35        _lzSendParam.sendParam.amountLD,
36        _lzSendParam.sendParam.minAmountLD,
37        _lzSendParam.sendParam.dstEid
38    );
39
40    // @dev Builds the options and OFT message to quote in the endpoint.
41    (bytes memory message, bytes memory options) =
42        _buildOFTMsgAndOptions(_lzSendParam.sendParam, _lzSendParam.
              extraOptions, _composeMsg, amountToCreditLD);
43
44    // @dev Sends the message to the LayerZero endpoint and returns the
          LayerZero msg receipt.
45    msgReceipt =
46        _lzSend(_lzSendParam.sendParam.dstEid, message, options,
              _lzSendParam.fee, _lzSendParam.refundAddress);
47    // @dev Formulate the OFT receipt.
48    oftReceipt = OFTReceipt(amountDebitedLD, amountToCreditLD);
49
50    emit OFTSent(msgReceipt.guid, _lzSendParam.sendParam.dstEid, msg.
          sender, amountDebitedLD, amountToCreditLD);
51 }
```

On the other hand, `IOftSender(tapOft).sendPacket(msg_.lzSendParams, "");` doesn't allow to pass any value along the call, i.e. `{value: msg.value}`.

So there is no way of paying the `lzSend` and this will simply revert.

Recommendation

Forward the `msg.value` passed along the lzCompose call.

```
1  -     IOftSender(tapOft).sendPacket(msg_.lzSendParams, "");
2  +     IOftSender(tapOft).sendPacket{value: msg.value}(msg_.lzSendParams,
       "");
```

[M-05] `UsdoMarketReceiverModule.lendOrRepayReceiver` repay flow is non-functional

Context

- UsdoMarketReceiverModule.sol#L78-L99

Description

`UsdoMarketReceiverModule.lendOrRepayReceiver` flow grants a few approvals before executing `_repay` or `_lend` actions:

```
1  function lendOrRepayReceiver(address srcChainSender, bytes memory _data
       ) public payable {
2      MarketLendOrRepayMsg memory msg_ = UsdoMsgCodec.
           decodeMarketLendOrRepayMsg(_data);
3
4      /**
5      * @dev validate data
6      */
7      msg_ = _validateLendOrRepayReceiver(msg_);
8
9      /**
10     * @dev Pearlmit approvals
11     */
12     // approve(address(msg_.lendParams.magnetar), msg_.lendParams.
           depositAmount);
13 >>>    approve(address(pearlmit), msg_.lendParams.depositAmount);
14 >>>    pearlmit.approve(
15         address(this),
16         0,
17         msg_.lendParams.magnetar,
18         uint200(msg_.lendParams.depositAmount),
19         uint48(block.timestamp + 1)
20     );
```

- For `approve(address(pearlmit), msg_.lendParams.depositAmount);` msg.sender is the `lzEndpoint` as this is called as part of `lzCompose`. So, it's `lzEndpoint` granting allowances to `Pearlmit` which is useless.
- `pearlmit.approve(...)` sets the allowance to pearlmit whereby `owner = TOFT`, `token = TOFT` and `operator = Magnetar`. In summary, `TOFT` is giving the approval to Magnetar to use it's own tokens.

If we check the logic downstream in the `MagnetarAssetModule.sol`:

```
## MagnetarAssetModule.sol

 function depositRepayAndRemoveCollateralFromMarket(
    DepositRepayAndRemoveCollateralFromMarketData memory data)
        public
        payable
    {
        /**
         * @dev validate data
         */
        _validateDepositRepayAndRemoveCollateralFromMarketData(data);

        IMarket _market = IMarket(data.market);
        IYieldBox _yieldBox = IYieldBox(_market._yieldBox());

        /**
         * @dev YieldBox approvals
         */
        _processYieldBoxApprovals(_yieldBox, data.market, true);

        /**
         * @dev deposit `market._assetId()` to YieldBox
         */
        if (data.depositAmount > 0) {
            uint256 assetId = _market._assetId();
            (, address assetAddress,,) = _yieldBox.assets(assetId);
>>>             data.depositAmount = _extractTokens(data.user,
   assetAddress, data.depositAmount);
            _depositToYb(_yieldBox, data.user, assetId, data.
                depositAmount);
        }

function _extractTokens(address _from, address _token, uint256 _amount)
     internal returns (uint256) {
    uint256 balanceBefore = IERC20(_token).balanceOf(address(this));
    // IERC20(_token).safeTransferFrom(_from, address(this), _amount);
>>>     bool isErr = pearlmit.transferFromERC20(_from, address(this),
   address(_token), _amount);
    if (isErr) revert Magnetar_ExtractTokenFail();
    uint256 balanceAfter = IERC20(_token).balanceOf(address(this));
    if (balanceAfter <= balanceBefore) revert Magnetar_ExtractTokenFail
        ();
    return balanceAfter - balanceBefore;
}
```

Tokens are extracted from the user through the `Pearlmit` with the `transferFrom` function on the TOFT.

```
## PermitC.sol

```

```
function _transferFromERC20(
        address token,
        address owner,
        address to,
        uint256 /*id*/,
        uint256 amount
      ) internal returns (bool isError) {
        isError = _beforeTransferFrom(token, owner, to, ZERO, amount);

        if (!isError) {
>>>             (bool success, bytes memory data) = token.call(abi.
    encodeWithSelector(IERC20.transferFrom.selector, owner, to, amount))
    ;
            if (!success) {
                isError = true;
            } else if (data.length > 0) {
                isError = abi.decode(data, (bool)) == false;
            }

            if (!isError) {
                isError =   _afterTransferFrom(token, owner, to, ZERO,
                    amount);
            }
        }
    }
```

In order for this to work: - The user needs to give Magnetar the approval to take his `TOFT` tokens through `Pearlmit`. - The user needs to give Pearlmit the allowance to spend his `TOFT` tokens as well.

The `_lend` logic part of the flow also expects several approvals downstream and suffers from the same issues described above.

Recommendation

Remove the unnecessary approvals and implement the necessary changes in all the contracts downstream to make the flow functional.

[M-06] Incorrect decoding in `decodeLockTwpTapDstMsg`

Context

- TapTokenCodec.sol#L62-L81

Description

During the Code4rena competitive audit, the following ***issue has been found***. It hasn't been fixed in the commit this audit points to.

Recommendation

Implement the changes recommended in the report.

[M-07] Adding reward tokens to `twTAP` results in `lzCompose` call permanently reverting

Context

- TapTokenReceiver.sol#L164-L169

Description

During the Code4rena competitive audit, the ***following issue has been found***.

It highlights a problem that in between a user submitting cross-chain call from `chainA` to `chainB`, additional rewards can be added in the `twTAP` contract. This makes `_claimTwpTapRewardsReceiver` revert due to the following check:

```
## TapTokenReceiver.sol

uint256[] memory claimedAmount_ = twTap.claimRewards(
    claimTwTapRewardsMsg_.tokenId, address(this));

// Check if the claimed amount is equal to the amount of sendParam
if (
    (claimedAmount_.length - 1) // Remove 1 because the first index
        doesn't count.
        != claimTwTapRewardsMsg_.sendParam.length
) {
    revert InvalidSendParamLength(claimedAmount_.length,
        claimTwTapRewardsMsg_.sendParam.length);
}

## TwTAP.sol

```

```
function addRewardToken(IERC20 _token) external onlyOwner returns (uint256) {
    if (rewardTokenIndex[_token] != 0) revert Registered();
    if (rewardTokens.length + 1 > maxRewardTokens) {
        revert TokenLimitReached();
    }
    rewardTokens.push(_token);

    uint256 newTokenIndex = rewardTokens.length - 1;
    rewardTokenIndex[_token] = newTokenIndex;

    emit AddRewardToken(address(_token), newTokenIndex);

    return newTokenIndex;
}
```

However, the report has overinflated severity as the claim that TapOFT tokens are going to be lost is incorrect.

```
function sendPacket(LZSendParam calldata _lzSendParam, bytes calldata _composeMsg)
    external
    payable
    returns (MessagingReceipt memory msgReceipt, OFTReceipt memory oftReceipt)
{
    // @dev Applies the token transfers regarding this send() operation.
    // - amountDebitedLD is the amount in local decimals that was ACTUALLY debited from the sender.
    // - amountToCreditLD is the amount in local decimals that will be credited to the recipient on the remote OFT instance.
    (uint256 amountDebitedLD, uint256 amountToCreditLD) = _debit(
        msg.sender,
        _lzSendParam.sendParam.amountLD,
        _lzSendParam.sendParam.minAmountLD,
        _lzSendParam.sendParam.dstEid
    );

    // @dev Builds the options and OFT message to quote in the endpoint.
    (bytes memory message, bytes memory options) =
        _buildOFTMsgAndOptions(_lzSendParam.sendParam, _lzSendParam.extraOptions, _composeMsg, amountToCreditLD);

    // @dev Sends the message to the LayerZero endpoint and returns the LayerZero msg receipt.
    msgReceipt =
        _lzSend(_lzSendParam.sendParam.dstEid, message, options, _lzSendParam.fee, _lzSendParam.refundAddress);
    // @dev Formulate the OFT receipt.
```

```
24      oftReceipt = OFTReceipt(amountDebitedLD, amountToCreditLD);
25
26      emit OFTSent(msgReceipt.guid, _lzSendParam.sendParam.dstEid, msg.
            sender, amountDebitedLD, amountToCreditLD);
27  }
```

LayerZero calls are split into `lzReceive` and `lzCompose` function invocations on the receiving chain. As `sendPacket` debits the OFT from user on the sending chain it credits the same amount in `lzReceive` on the receiving chain.

```
1   function _lzReceive(
2       Origin calldata _origin,
3       bytes32 _guid,
4       bytes calldata _message,
5       address, /*_executor*/ // @dev unused in the default implementation
            .
6       bytes calldata /*_extraData*/ // @dev unused in the default
            implementation.
7   ) internal virtual override {
8       // @dev The src sending chain doesn't know the address length on
            this chain (potentially non-evm)
9       // Thus everything is bytes32() encoded in flight.
10      address toAddress = _message.sendTo().bytes32ToAddress();
11      // @dev Convert the amount to credit into local decimals.
12      uint256 amountToCreditLD = _toLD(_message.amountSD());
13      // @dev Credit the amount to the recipient and return the ACTUAL
            amount the recipient received in local decimals
14  >>>       uint256 amountReceivedLD = _credit(toAddress,
        amountToCreditLD, _origin.srcEid);
15
16      if (_message.isComposed()) {
17          // @dev Stores the lzCompose payload that will be executed in a
                separate tx.
18          // Standardizes functionality for executing arbitrary contract
                invocation on some non-evm chains.
19          // @dev The off-chain executor will listen and process the msg
                based on the src-chain-callers compose options passed.
20          // @dev The index is used when a OApp needs to compose multiple
                msgs on lzReceive.
21          // For default OFT implementation there is only 1 compose msg
                per lzReceive, thus its always 0.
22          endpoint.sendCompose(
23              address(this), // Updated from default `toAddress`
24              _guid,
25              0, /* the index of the composed message*/
26              _message.composeMsg()
27          );
28      }
29
30      emit OFTReceived(_guid, _origin.srcEid, toAddress, amountReceivedLD
```

```
            );
31  }
```

Any message set in the `composeMsg` will get executed in a separate transaction as part of the `lzCompose` flow. In the case of `_claimTwpTapRewardsReceiver` if it permanently reverts it will cause the loss of all `msg.value` paid for sending rewards with `sendPacket`.

```
1  ## TapTokenReceiver.sol
2
3   // Send back packet
4  TapTokenSender(rewardToken_).sendPacket{
5  >>>     value: claimTwTapRewardsMsg_.sendParam[sendParamIndex].fee.
       nativeFee
6  }(claimTwTapRewardsMsg_.sendParam[sendParamIndex], bytes(""));
```

Recommendation

Consider pausing the `sendPacket` functionality during governance updates on the `twTAP` contract, and then waiting enough time for any potential pending transactions to finish processing and only then execute governance actions, e.g. `addRewardToken`.

[L-01] PoolIds should be required for all the connectedOFTs

Context

- Balancer.sol#L252

Description

`initConnectedOFT` function allows to initialize a source and destination OFT-related data with the corresponding pool ids. The function logic requires passing the src and dst pool ids in case the `srcOFT != native`.

```
1  ## Balancer.sol
2
3   >> if (!isNative && _ercData.length == 0) revert PoolInfoRequired();
4
5  (uint256 _srcPoolId, uint256 _dstPoolId) = abi.decode(_ercData, (
       uint256, uint256));
```

It should be required to pass the pool ids in both cases as `srcPoolId` is always required to fetch the conversion rate.

```
## Balancer.sol

address stargatePool = stargateFactory.getPool(connectedOFTs[_srcOft][
    _dstChainId].srcPoolId);
```

Recommendation

```
-         bool isNative = ITOFT(_srcOft).erc20() == address(0);
-         if (!isNative && _ercData.length == 0) revert PoolInfoRequired
    ();
+         if (_ercData.length == 0) revert PoolInfoRequired();
```

[L-02] marketHelper is not whitelisted in removeAssetReceiver flow

Context

- UsdoMarketReceiverModule.sol#L223-L225

Description

`removeAssetReceiver` flow checks the whitelist status of multiple (Magnetar, Singularity, BigBang) contracts but doesn't check if the `marketHelper` contract is whitelisted. Although there is a whitelist check downstream inside the `MagnetarOptionModule`, it is advised to do the check in the `UsdoMarketReceiverModule` as well, in case logic in the `Magnetar` changes.

Recommendation

Add `_checkWhitelistStatus` for `marketHelper` contract inside the `UsdoMarketReceiverModule.removeAssetReceiver` flow.