Tapioca Security Review

Auditors

Dimiitria, Lead Security Researcher

Rajeev, Lead Security Researcher

Alex The Entreprenerd, Security Researcher

Calvin Boehr, Junior Security Researcher

Hrishibat, Junior Security Researcher

Report prepared by: Lucas Goiriz

February 19, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

TapiocaDAO is a decentralized autonomous organization (DAO) represented by a Cayman Islands Foundation, creating an ecosystem of protocols including the first-ever Omnichain money market & censorship resistant U.S. Dollar stablecoin across preeminent EVM networks, through leveraging the modular LayerZero generalized messaging network.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of tap-token, tapioca-bar and tapiocaZ according to the specific commits. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 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 - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

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

4 Executive Summary

Over the course of 45 days in total, Tapioca engaged with Spearbit to review the tap-token, Tapioca-bar, TapiocaZ protocol. In this period of time a total of **194** issues were found.

Summary

Project Name	Tapioca
Repository	tap-token, Tapioca-bar, TapiocaZ
Commit	75688e...952d8e, f15aa5...388bdb, 9ef97e...492b08
Type of Project	DeFi, Omnichain Swaps
Audit Timeline	Nov 28 to Jan 29

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	7	3	0
High Risk	31	12	3
Medium Risk	57	11	10
Low Risk	59	4	9
Gas Optimizations	11	3	4
Informational	29	4	8
Total	**194**	**37**	**34**

Note: Given that the cross-chain TOFT strategy flows will be removed, the review team has not looked into `BaseTOFTStrategyModule` and `BaseTOFTStrategyDestinationModule`.

5 Findings

5.1 Critical Risk

5.1.1 `participate()` can be run late, stealing from other lockers due to not accounting for attacker's lock shares in `netDepositedForEpoch[epoch]`

Severity: Critical Risk

Context: oTAP.sol#L118-L134, oTAP.sol#L39-L43, TapiocaOptionBroker.sol#L459-L465

Description: There is no control on when `participate()` is run after `lock()`, while `participate()` accounts for lock's shares starting from the next epoch, assuming that it is run simultaneously with the `lock()`. Ability to exercise meanwhile is based only on the properties of the lock itself. This way an attacker can `lock()`, wait (for example for one `EPOCH_DURATION`), then `participate()` and immediately `exerciseOption()`. They will receive outsized payoff as option payoff base, `netDepositedForEpoch`, won't account for this lock's shares in the current epoch, while the option can be exercised in this epoch.

`netDepositedForEpoch` is being increased in `participate()` starting with `epoch + 1` period, assuming that `lock()` was run in the current epoch (see TapiocaOptionBroker.sol#L350-L360):

```
// Record amount for next epoch exercise
// see the line below
netDepositedForEpoch[epoch + 1][lock.sglAssetID] += int256(
    uint256(lock.ybShares)
);

uint256 lastEpoch = _timestampToWeek(lock.lockTime + lock.lockDuration);
// And remove it from last epoch
// Math is safe, check `_emitToGauges()`
netDepositedForEpoch[lastEpoch + 1][lock.sglAssetID] -= int256(
    uint256(lock.ybShares)
);
```


But there is no control for that, an attacker can run `participate()` after a while, as long as `epoch <= ((_lock.lockTime + _lock.lockDuration - emissionsStartTime) / WEEK)` and `_isPositionActive(lock) == true` this way (see TapiocaOptionBroker.sol#L280-L288):

```
function participate(
    uint256 _tOLPTokenID
) external whenNotPaused returns (uint256 oTAPTokenID) {
    // Compute option parameters
    LockPosition memory lock = tOLP.getLock(_tOLPTokenID);
    bool isPositionActive = _isPositionActive(lock);
    if (!isPositionActive) revert OptionExpired();

    if (lock.lockDuration < EPOCH_DURATION) revert DurationTooShort();
```

Exercise is then checks the `lock.lockTime` only, so exercising is possible whenever `block.timestamp >= lock.lockTime + EPOCH_DURATION` (see TapiocaOptionBroker.sol#L442-L449):

```
function exerciseOption(
    uint256 _oTAPTokenID,
    ERC20 _paymentToken,
    uint256 _tapAmount
) external whenNotPaused {
    // Load data
    (, TapOption memory oTAPPosition) = oTAP.attributes(_oTAPTokenID);
    // see the line below
    LockPosition memory tOLPLockPosition = tOLP.getLock(oTAPPosition.tOLP);
```

Also see TapiocaOptionBroker.sol#L459-L465:

```
// Check requirements
if (paymentTokenOracle.oracle == IOracle(address(0)))
    revert PaymentTokenNotSupported();
if (!oTAP.isApprovedOrOwner(msg.sender, _oTAPTokenID))
    revert NotAuthorized();
// see the line below
if (block.timestamp < tOLPLockPosition.lockTime + EPOCH_DURATION)
    revert OneEpochCooldown(); // Can only exercise after 1 epoch duration
```

So the attacker can `lock()`, wait for one `EPOCH_DURATION`, call `participate()` → `exerciseOption()`, and receive an outsized payoff. The attack can be tuned so that this happens in the very beginning of the epoch and the attacker steals the substantial chunk of the epoch rewards. This can be repeated for each epoch with the help of array of positions.

Impact: the attacker can skip the `netDepositedForEpoch` accounting for their shares in the current epoch, increasing the option payoff base for all lockers in the epoch, making the system insolvent in the epoch, and receiving more on exercise. Due to insolvency that follows from `sum(shares) > netDepositedForEpoch[epoch]` state the attacker need to exercise early, so there are some additional MEV costs, which, however, are far surpassed by the expected payoff. As an example, if the attacker's shares are exactly the half of all shares in the system for the epoch, they will be able to retrieve all rewards for themselves as it will be `lock.ybShares == netDepositedForEpoch[epoch][lock.sglAssetID]`.

Likelihood: High (no low probability prerequisites, only some MEV costs to be the first to run exercise) + Impact: High (substantial part of epoch rewards can be stolen, system insolvency) = Severity: Critical.

Recommendation: Consider basing the exercise delay on the participation timestamp instead of the lock timestamp, e.g. (oTAP.sol#L118-L134):


```
  function mint(
      address _to,
      uint128 _expiry,
      uint128 _discount,
      uint256 _tOLP
  ) external returns (uint256 tokenId) {
      require(msg.sender == broker, "OTAP: only onlyBroker");
      tokenId = ++mintedOTAP;
      _safeMint(_to, tokenId);

      TapOption storage option = options[tokenId];
+     option.entry = block.timestamp;
      option.expiry = _expiry;
      option.discount = _discount;
      option.tOLP = _tOLP;

      emit Mint(_to, tokenId, option);
  }
```

And oTAP.sol#L39-L43:

```
  struct TapOption {
+     uint128 entry;  // time when the option position was created
      uint128 expiry; // timestamp, as once one wise man said, the sun will go dark before this
↪   overflows
      uint128 discount; // discount in basis points
      uint256 tOLP; // tOLP token ID
  }
```

And finally TapiocaOptionBroker.sol#L459-L465:

```
  // Check requirements
  if (paymentTokenOracle.oracle == IOracle(address(0)))
      revert PaymentTokenNotSupported();
  if (!oTAP.isApprovedOrOwner(msg.sender, _oTAPTokenID))
      revert NotAuthorized();
- if (block.timestamp < tOLPLockPosition.lockTime + EPOCH_DURATION)
+ if (block.timestamp < oTAPPosition.entry + EPOCH_DURATION)
      revert OneEpochCooldown(); // Can only exercise after 1 epoch duration
```

Tapioca: Fixed in PR 162.

Spearbit: Fix looks ok.

5.1.2 `TapiocaOptionBroker`'s `participate()` and `exitPosition()` can be iterated in one block to drive `pool.averageMagnitude` arbitrarily low and steal all rewards for the epoch

Severity: Critical Risk

Context: TapiocaOptionBroker.sol#L280-L299

Description: When `block.timestamp == lock.lockTime + lock.lockDuration`, it is possible to run both `participate()` and `exitPosition()`. Iterating these calls will drive `netDepositedForEpoch[epoch + 1][lock.sglAssetID]` up, `netDepositedForEpoch[_timestampToWeek(lock.lockTime + lock.lockDuration) + 1][lock.sglAssetID]` and `pool.averageMagnitude` down. `netDepositedForEpoch` will be netted thereafter with `newEpoch()` → `_emitToGauges()` calls, but `pool.averageMagnitude` value will persist. Such calls can be made during the block with a specific timestamp only, but in order to move `pool.averageMagnitude` there is no need to front-run anything and the gas costs will be usual in this case, the only limit on the number of `participate()` → `exitPosition()` iterations is the block gas limit. Also, there is no need to forfeit option right for the position that is operated as this manipulation cycle can happen right after the usual `participate()` → `exerciseOption()` → `exitPosition()` calls were executed and option payoff received.

This way, an attacker can cheaply manipulate `pool.averageMagnitude` downwards arbitrarily as this can be performed from many accounts, each locked for the minimum `EPOCH_DURATION`. There are no costs above usual gas spending.

Schematic proof of concept:

1. Run `lock()` with `lock.lockDuration == EPOCH_DURATION`.
2. After exactly `EPOCH_DURATION`, in the block with `block.timestamp == lock.lockTime + lock.lockDuration` run `participate()` → `exerciseOption()` → `exitPosition()` to retrieve option rewards (`tx1`).
3. In the same block atomically run many, as remaining block gas limit allows, iterations of `participate()` → `exitPosition()`, i.e. say {20 x (`participate()`, `exitPosition()`)} (`tx2`).

The position needs to meet the `lock.ybShares >= computeMinWeight(pool.totalDeposited, MIN_WEIGHT_FACTOR)` condition to have `hasVotingPower == true`, but since the option payoff for the position is preserved, this does not add to the costs of the attack, which can be attached to a normal workflow of any big enough position.

The result is that `pool.averageMagnitude` can be arbitrary manipulated downwards, so the system adjustment capability be substantially reduced.

It will be `block.timestamp == tOLPLockPosition.lockTime + EPOCH_DURATION` in (2) and (3) and `EPOCH_DURATION` since locking, `_isPositionActive()` in (3) will be true as `epoch <= _timestampToWeek(block.timestamp) == _timestampToWeek(_lock.lockTime + _lock.lockDuration)`:

- TapiocaOptionBroker.sol#L620-L631:

```
function _isPositionActive(
    LockPosition memory _lock
) internal view returns (bool isPositionActive) {
    if (_lock.lockTime <= 0) revert PositionNotValid();
    if (_isSGLInRescueMode(_lock)) revert SingularityInRescueMode();

    uint256 expiryWeek = _timestampToWeek(
        _lock.lockTime + _lock.lockDuration
    );

    isPositionActive = epoch <= expiryWeek;
}
```

While `exitPosition()` doesn't modify `pool.averageMagnitude`, `participate()` reduces it each time with `pool.totalParticipants` division (it is not an average, but rather a step, so reducing it removes the flexibility from the system, `pool.cumulative` will become mostly constant afterwards):

- TapiocaOptionBroker.sol#L314-L318:

```
if (hasVotingPower) {
    pool.totalParticipants++; // Save participation
    pool.averageMagnitude =
        (pool.averageMagnitude + magnitude) /
        pool.totalParticipants; // compute new average magnitude
```

This can be used, as an example, to lock low `pool.cumulative` value, which will supply substantial discounts (the product of `computeTarget()`) for low enough `_timeWeight = lock.lockDuration` on a constant basis after the attack:

- TapiocaOptionBroker.sol#L301-L305:

```
uint256 magnitude = computeMagnitude(
    uint256(lock.lockDuration),
    pool.cumulative
);
uint256 target = computeTarget(dMIN, dMAX, magnitude, pool.cumulative);
```

- twAML.sol#L121-L142:

```
function computeMagnitude(
    uint256 _timeWeight,
    uint256 _cumulative
) internal pure returns (uint256) {
    return
        sqrt(_timeWeight * _timeWeight + _cumulative * _cumulative) -
        _cumulative;
}

function computeTarget(
    uint256 _dMin,
    uint256 _dMax,
    uint256 _magnitude,
    uint256 _cumulative
) internal pure returns (uint256) {
    if (_cumulative == 0) {
        return _dMax;
    }
    uint256 target = (_magnitude * _dMax) / _cumulative;
    target = target > _dMax ? _dMax : target < _dMin ? _dMin : target;
    return target;
}
```

- TapiocaOptionBroker.sol#L362-L368:

```
// Mint oTAP position
oTAPTokenID = oTAP.mint(
    msg.sender,
    lock.lockTime + lock.lockDuration,
    uint128(target),
    _tOLPTokenID
);
```

Impact: running `pool.averageMagnitude` arbitrarily downwards, so that the system can no longer adjust itself thereafter. This can be used to inflate overall TAP emission by locking small pool `cumulative`, so that low duration locks will permanently have substantial discounts. This can be the direct goal of attackers, but it also decreases long term protocol owned liquidity as higher discounts mean both higher emissions and lower costs for the lockers. The total cost is normal gas price of `tx2` in (3) step, which is negligible in L2 networks.

Likelihood: High (low cost far less than the potential benefits provides high overall probability) + Impact: High (permanent high emission is detrimental for TAP value, being equivalent to stealing from existing TAP holders) = Severity: Critical.

Also, the same surface enables stealing of the entire epoch rewards as `participate()` → `exerciseOption()` → `exitPosition()` can be iterated in the same setup as well (this attack was proposed in C4-321).

This is possible as `exerciseOption()` is based on the `oTAPCalls[_oTAPTokenID]` accounting, while oTAP token is minted in `participate()` and then burned in `exitPosition()`:

- TapiocaOptionBroker.sol#L362-L368:

```
// Mint oTAP position
oTAPTokenID = oTAP.mint(
    msg.sender,
    lock.lockTime + lock.lockDuration,
    uint128(target),
    _tOLPTokenID
);
```

- TapiocaOptionBroker.sol#L427-L430:

```
// Delete participation and burn oTAP position
address otapOwner = oTAP.ownerOf(_oTAPTokenID);
delete participants[oTAPPosition.tOLP];
oTAP.burn(_oTAPTokenID); // See here
```

So `_oTAPTokenID` is renewed with each iteration, that allows to bypass the `oTAPCalls` based check as `oTAPCalls[_oTAPTokenID][cachedEpoch]` is zero each time with new `_oTAPTokenID`:

- TapiocaOptionBroker.sol#L475-L485:

```
uint256 eligibleTapAmount = muldiv(
    tOLPLockPosition.ybShares,
    gaugeTotalForEpoch,
    netAmount
);
// see the two lines below
eligibleTapAmount -= oTAPCalls[_oTAPTokenID][cachedEpoch]; // Subtract already exercised amount
if (eligibleTapAmount < _tapAmount) revert TooHigh();

uint256 chosenAmount = _tapAmount == 0 ? eligibleTapAmount : _tapAmount;
if (chosenAmount < 1e18) revert TooLow();
oTAPCalls[_oTAPTokenID][cachedEpoch] += chosenAmount; // Adds up exercised amount to current
↪  epoch
```

Impact: this allows for exercising the same position multiple times, until all the epoch rewards be extracted. It has some MEV cost as front running other participants is needed, but payoff far surpasses such costs. The rewards can be stolen from every epoch with the help of sequence of locks (which can be reused for that matter) and positions.

Likelihood: High (low cost far less than the potential benefits provides high overall probability) + Impact: High (all rewards can be stolen in each epoch) = Severity: Critical.

Note, that the ability to exercise immediately right after `participate()` for a week old lock described in the issue "Participate can be run late, stealing from other lockers due to not accounting for attacker's lock shares in `netDepositedForEpoch[epoch]`" enables this attack as well. Both surfaces look to be closed by the joint recommended mitigation for these issues.

Recommendation: The root issue is the ability to perform entry and exit at the same moment of time. Consider making `participate()` and `exitPosition()` conditions mutually exclusive, e.g.:

- TapiocaOptionBroker.sol#L280-L288:

```
    function participate(
        uint256 _tOLPTokenID
    ) external whenNotPaused returns (uint256 oTAPTokenID) {
        // Compute option parameters
        LockPosition memory lock = tOLP.getLock(_tOLPTokenID);
+       if (block.timestamp >= lock.lockTime + lock.lockDuration)
+           revert LockExpired();
        bool isPositionActive = _isPositionActive(lock);
        if (!isPositionActive) revert OptionExpired();

        if (lock.lockDuration < EPOCH_DURATION) revert DurationTooShort();
```

Tapioca: Fixed in PR 161.

Spearbit: Fix looks ok.

5.1.3 Anyone can bypass `USDOMarketModule.removeAsset()` checks to call `Magnetar.exitPositionAndRemoveCollateral()` and steal any user's tOLP token and underlying YB shares

Severity: Critical Risk

Context: USDOMarketModule.sol#L29-L67, USDOMarketDestinationModule.sol#L173-L209, MagnetarV2.sol#L807-L834, MagnetarV2Storage.sol#L243-L246, MagnetarMarketModule.sol#L540-L751

Description: `USDOMarketModule.removeAsset()` accepts `from` and `to` addresses and performs allowance checks on `from` if `from != msg.sender`. However, `from` is not used thereafter while `to` is encoded and sent to the destination chain to be used as the `user` argument for `Magnetar.exitPositionAndRemoveCollateral()`. This means that anyone can set `from == msg.sender` to bypass the allowance checks and call `Magnetar.exitPositionAndRemoveCollateral()` for any user `to` on the destination chain.

`Magnetar.exitPositionAndRemoveCollateral()` is meant to be a "*helper to exit from tOB, unlock from tOLP, remove from SGL, repay on BB, remove collateral from BB and withdraw*" as documented. It enforces an access control check using `_checkSender(user)` which checks if `_from == msg.sender || cluster.isWhitelisted(0, msg.sender)`. While Magnetar is OOS for this review, it appears that `exitPositionAndRemoveCollateral()` is typically meant to be called by users to operate on their protocol positions. The cluster whitelist check allows `USDOMarketDestinationModule` to execute this function on the user `to` as triggered from the source chain.

However, another security review which happened in parallel on the `tapioca-periph` repository which had `Magnetar` in-scope raised a related Critical issue C-03 which found that `Magnetar.exitPositionAndRemoveCollateral()` may be used to steal the user's tOLP token and underlying YB shares by exploiting prior approvals.

Impact: Anyone can bypass `USDOMarketModule.removeAsset()` checks to call `Magnetar.exitPositionAndRemoveCollateral()` for any user `to` on the destination chain which affects

their oTAP, tOLP, SGL or BB positions depending on their approvals and may lead to unexpected outcomes deviating from specified behavior. Specifically, they can steal any user's tOLP token and underlying YB shares on the destination chain.

Likelihood: High + Impact: High (Loss of tOLP token and underlying liquidity) = Severity: Critical.

Recommendation:

1. Revisit the rationale for having a `from` parameter in `USDOMarketModule.removeAsset()`. Consider removing that in favor of having a single `user` parameter (effectively the same as the current `to` parameter) on which allowance checks are performed on the source chain and which is acted upon by `Magnetar.exitPositionAndRemoveCollateral()`.
2. Mitigate C-03 from the other security review.

Tapioca: Addressed in PR 322.

5.1.4 BBLiquidation's and SGLLiquidation's `_updateBorrowAndCollateralShare()` mix up base and elastic units, limits full liquidation amount to the current debt, can fully remove liquidator incentive, which can block valid liquidations and endangers protocol health

Severity: Critical Risk

Context: BBLiquidation.sol#L156-L215, SGLLiquidation.sol#L186-L245

Description: The list of issues with `_updateBorrowAndCollateralShare()`, gathered here to ease the cumulative mitigation:

1. `maxBorrowParts` and `userBorrowPart`, being in `part` units, are compared to the current, being in `elastic` units, `borrowPartWithBonus` derived amounts across the logic. These two bases represent as of inception and current units of accounting correspondingly and can be drastically different: while `part` is a fixed point of time, `elastic` drifts away from it with accruals and the accumulated difference can have orders of magnitude. This is case 2 of the issue "Incorrect application of elastic units across Liquidations".

   It is high probability issue being the part of basic liquidation workflow and having no material prerequisites. These are a variety of cases, share of that will grow along with growing `elastic / base` inflator over time, in which valid liquidations will be blocked due to the logic.

   Likelihood: High + Impact: High = Severity: Critical.

2. Total liquidation amount is limited to fit into the current debt, which leads to incentivizing liquidators to leave some remainder of the principal in order to keep their liquidation bonus intact. This way the logic forces all liquidations that exceed the current borrower's balance to be handled by manual `liquidateBadDebt()`.

   Each liquidation then will be partial: no liquidator will agree to remove liquidation bonus fully, so they will supply only partial amounts to fit the `borrowPartWithBonus <= elastic(userBorrowPart[user])` restriction, while still receiving max bonus. This way there always will be some remainder amount left in such loans, that will accumulate and be left for manual treatment by an owner, which may be manageable at the early stages, but can quickly go out of hand along with increased loan count and sharp market movements. Timely liquidations are important for overall system stability as they control the risk of USDO becoming undercollateralized and its depeg.

   Similarly, it's a high probability issue, there is no material prerequisites. Impact is lower as these remainder amounts will take some time to accumulate.

   Likelihood: High + Impact: Medium = Severity: High.

3. The `collateralPartInAsset == borrowPartWithBonus` state imply perfect liquidation bonus coverage and there is no need to revert with `BadDebt()` in this case. It might be `collateralShare > userCollateralShare[user]` due to rounding and still revert with `NotEnoughCollateral()`, but that's the same outcome and the liquidator can rerun with reduced amount if this happens. If `collateralShare == userCollateralShare[user]` there are no issues and liquidation can conclude.

   Medium probability as this state will be reached from time to time only, while the impact is similar to (2).

Likelihood: Medium + Impact: Medium = Severity: Medium.

4. The checks for max amounts, both user balance and `maxBorrowParts` wise, happen independently, this way if for after liquidator bonus amount a check triggers, it will remove liquidator bonus completely, i.e. liquidator can lose all the incentives. This will force the liquidators to hold off the actions, which will worsen the stability of the system and increase overall risk of undercollateralization and depeg. Also, what liquidator wants to control is max USDO they have to supply, `borrowAmount`, but it is based on `borrowPart` only, not `borrowPartWithBonus`, so there is no need to control the latter for this limit.

   It's a follow-up consideration of (2). Since `maxBorrowParts` is controlled by liquidator it's more manageable than check vs balance (2) describes, so the probability this limiting the operations is lower, can be estimated as medium, while impact is same as in (2). Overall (2) and (4) mean that the limitation checks need to be applied to the pre bonus amount only.

   Likelihood: Medium + Impact: Medium = Severity: Medium.

5. Solvency check is now done in the very end, while since `borrowPart` is the least amount in question, when it is zero the derived ones also will be zeros, so it is possible and recommended to check for solvency earlier to save gas.

   Severity: Gas Optimization.

Recommendation: Per list above:

1. Argument description for `liquidate()` can be updated to indicate that `maxBorrowParts` are in current, `elastic`, amounts. `maxBorrowParts` look to be used only here, so an alternative is to convert `maxBorrowParts` to elastic, but it can be somewhat more convenient for liquidators to set current amounts directly instead of more abstract `part` values. User balance, `userBorrowPart`, needs to be converted to elastic units before comparison.
2. Consider allowing liquidation incentive to increase the user debt, i.e. introducing the logic where the total with bonus included can go above current debt as long as collateral allows so. The willingness to receive less than a full liquidation bonus can be an additional argument to `liquidate()` function, say `minLiquidationBonus` representing min accepted bonus by the liquidator instead of `liquidationBonusAmount` (having the same `FEE_PRECISION`). The absence of ability to provide any bonus to be treated as a bad debt case and should lead to revert.
3. Consider allowing the equality case, e.g.:

```
- if (collateralPartInAsset <= borrowPartWithBonus) revert BadDebt();
+ if (collateralPartInAsset < borrowPartWithBonus) revert BadDebt();
```

4. Since `borrowPart` is deemed to be `borrowPartWithBonus` without the bonus part, derive one from another after the checks, so this relationship be ensured. Structure checks so that `borrowPart` is controlled to be below both user balance (there is no need and possibility to repay more) and liquidator supplied limit. Recalling (2), we have that `borrowPartWithBonus` doesn't have to be within either limit as bonus can go above balance (2), while liquidator limit is meant to be for USDO they have to pay, not for the total `borrowPartWithBonus`.
5. Solvency check can be done right after `borrowPart` is calculated (not earlier in order to keep rounding impact in scope for this check). Combining these together it can be implemented as follows, as an example:
   - BBLiquidation.sol#L156-L215

```
    function _updateBorrowAndCollateralShare(
        address user,
        uint256 maxBorrowPart,
+       uint256 minLiquidationBonus,  // min liquidation bonus to accept, with 0 for default
        uint256 _exchangeRate
    )
        private
        returns (
            uint256 borrowAmount,
            uint256 borrowPart,
            uint256 collateralShare
```

```
          )
      {
          if (_exchangeRate == 0) revert ExchangeRateNotValid();
          uint256 collateralPartInAsset = (yieldBox.toAmount(
              collateralId,
              userCollateralShare[user],
              false
          ) * EXCHANGE_RATE_PRECISION) / _exchangeRate;

          uint256 borrowPartWithBonus = computeClosingFactor(
              userBorrowPart[user],
              collateralPartInAsset,
              FEE_PRECISION_DECIMALS
          );

+         // limit liquidable amount before bonus to the current debt
+         uint256 userTotalBorrowAmount = totalBorrow.toElastic(userBorrowPart[user], true);
+         borrowPartWithBonus = borrowPartWithBonus > userTotalBorrowAmount
+             ? userTotalBorrowAmount
+             : borrowPartWithBonus;

+         // check the amount to be repaid versus liquidator supplied limit
+         borrowPartWithBonus = borrowPartWithBonus > maxBorrowPart
+                 ? maxBorrowPart
+                 : borrowPartWithBonus;
+
-         borrowPart = borrowPartWithBonus;
+         borrowAmount = borrowPartWithBonus;
+         // calculating part units, preventing rounding dust when liquidation is full
+         borrowPart = borrowAmount == userTotalBorrowAmount
+             ? userBorrowPart[user]
+             : totalBorrow.toBase(borrowPartWithBonus, false);
+         if (borrowPart == 0) revert Solvent();

          if (liquidationBonusAmount > 0) {
              borrowPartWithBonus =
                  borrowPartWithBonus +
                  (borrowPartWithBonus * liquidationBonusAmount) /
                  FEE_PRECISION;
          }

-         borrowPartWithBonus = maxBorrowPart > borrowPartWithBonus
-             ? borrowPartWithBonus
-             : maxBorrowPart;
-         borrowPartWithBonus = borrowPartWithBonus > userBorrowPart[user]
-             ? userBorrowPart[user]
-             : borrowPartWithBonus;

-         if (collateralPartInAsset <= borrowPartWithBonus) revert BadDebt();
+         if (collateralPartInAsset < borrowPartWithBonus) {
+             if (collateralPartInAsset <= userTotalBorrowAmount) revert BadDebt();
+             // If current debt is covered by collateral fully
+             // then there is some liquidation bonus,
+             // so liquidation can proceed if liquidator's minimum is met
+             if (minLiquidationBonus > 0) {
+                 // `collateralPartInAsset > borrowAmount` as `borrowAmount <=
↪  userTotalBorrowAmount`
+                 uint256 effectiveBonus = ((collateralPartInAsset - borrowAmount) *
↪  FEE_PRECISION) / borrowAmount;
+                 if (effectiveBonus < minLiquidationBonus) revert
↪  InsufficientLiquidationBonus();
+                 // borrowPartWithBonus = collateralPartInAsset;
```

```
+             collateralShare = userCollateralShare[user];
+         } else {
+             revert InsufficientLiquidationBonus();
+         }
+     } else {
+         collateralShare = yieldBox.toShare(
+             collateralId,
+            (borrowPartWithBonus * _exchangeRate) / EXCHANGE_RATE_PRECISION,
+             false
+         );

+         if (collateralShare > userCollateralShare[user])
+             revert NotEnoughCollateral();
+     }

-     borrowPart = maxBorrowPart > borrowPart ? borrowPart : maxBorrowPart;
-     borrowPart = borrowPart > userBorrowPart[user]
-         ? userBorrowPart[user]
-         : borrowPart;

      userBorrowPart[user] = userBorrowPart[user] - borrowPart;

-     borrowAmount = totalBorrow.toElastic(borrowPart, false);
-     collateralShare = yieldBox.toShare(
-         collateralId,
-         (borrowPartWithBonus * _exchangeRate) / EXCHANGE_RATE_PRECISION,
-         false
-     );

-     if (collateralShare > userCollateralShare[user])
-         revert NotEnoughCollateral();
      userCollateralShare[user] -= collateralShare;
-     if (borrowAmount == 0) revert Solvent();
```

- BBLiquidation.sol#L21

```
    error BadDebt();
+   error InsufficientLiquidationBonus();
```

This also to be ported to SGLLiquidation's `_updateBorrowAndCollateralShare()`, since the code there is the same.

Tapioca: Created PR 298.

Spearbit: Fix looks ok.

5.1.5 Missing nonce logic mints TapOFTs out-of-thin-air for cross-chain `twTap.participate()` when failed messages are retried

Severity: Critical Risk

Context: BaseTapOFT.sol#L147-L182, BaseTOFTStrategyDestinationModule.sol#L56-L60

Description: LayerZero documentation implies that nonce usage is only for message correlation. Their documentation on message passing implies that it is up to the application on the destination chain to manage any message-handling triggered logic-/EVM-level failures and future retries of the same. This means that wherever the destination user application (UA) expects message failures and therefore is responsible for storing them to enable future retries, it needs to make sure that the previously performed storage changes before the failures are: 1) Acceptable in that partially successful state 2) Do not result in double-spends or other odd behavior when the failed messages are retried. To mitigate (2), the destination UA needs to enforce nonce logic.

While the protocol enforces this in most required places on Destination Modules, it is missing in `BaseTapOFT._lockTwTapPosition()` which has `twTap.participate()` in a `try-catch` block to handle failures and performs `_storeFailedMessage()` for unexpected reverts, but does not implement nonce logic to prevent repeated `_creditTo(_srcChainId, address(this), amount);` on retries.

Impact: Tokens minted in `_creditTo()` on retries are not backed by an equivalent amount on the source chain because that is performed only once. Missing nonce logic in `BaseTapOFT._lockTwTapPosition()` for cross-chain `twTap.participate()` therefore mints TapOFTs tokens out-of-thin-air on destination chain when failed messages are retried.

Proof of Concept:

1. Call `lockTwTapPosition()` with incorrect `duration` (which will ensure a caught revert on destination chain).
2. `_lockTwTapPosition()` reverts with a Custom Error (not caught by the `string`).
3. Replay the `_storeFailedMessage` until all tokens are drainable.

Likelihood: High + Impact: High = Severity: Critical.

Recommendation: Add nonce and `creditedPackets` logic as shown below:

```
- _creditTo(_srcChainId, address(this), amount);
+ bool credited = creditedPackets[_srcChainId][_srcAddress][_nonce];
+ if (!credited) {
+     _creditTo(_srcChainId, address(this), amount);
+     creditedPackets[_srcChainId][_srcAddress][_nonce] = true;
+ }
```

and revisit all logic on destination chain where failures and retries are expected to consider similar issues for appropriate mitigation.

Note: A review of LayerZero base layer was out-of-scope for this effort and their integration guidelines are not very clear about these aspects.

5.1.6 Missing access control allows anyone to arbitrarily change whitelisted contracts and LayerZero chain ID

Severity: Critical Risk

Context: BaseUSDO.sol#L182-L188, BaseTOFT.sol#L521-L527, MagnetarV2.sol#L70-L77, Cluster.sol

Description: Tapioca uses the `Cluster` contract to manage the whitelist status of different contracts used across the protocol and update the LayerZero chain ID. While the Cluster contract address is set in different constructors, there is also a `setCluster()` setter defined in different components to update the Cluster contract address if required. This should clearly be a privileged operation.

While `Penrose.setCluster()` and `BaseLeverageExecutor.setCluster()` correctly enforce `onlyOwner` modifier access control, `BaseUSDO.setCluster()`, `BaseTOFT.setCluster()` and `MagnetarV2.setCluster()` (out of scope for this review) are missing this access control.

Impact: Anyone can update the Cluster contract address to arbitrarily change whitelisted contracts and LayerZero chain ID, which should allow them to subvert the entire protocol.

Likelihood: High + Impact: High = Severity: Critical

Recommendation: Add missing `onlyOwner` modifier access control on the identified `setCluster()` functions.

Tapioca: Fixed in PR 271.

Spearbit: Verified.

5.1.7 BigBang `borrow()` will always revert when protocol enforces `totalBorrowCap`

Severity: Critical Risk

Context: BBLendingCommon.sol#L65-L68, BBBorrow.sol#L48

Description: `BBLendingCommon._borrow()` will revert with `BorrowCapReached` when `totalBorrow.elastic <= totalBorrowCap` if `totalBorrowCap > 0`. Given that `totalBorrowCap` is expected to enforce an upper threshold cap on total borrow amount, this will revert even for the very first borrow. And it is very likely that the protocol will enforce `totalBorrowCap`.

This is due to the incorrect comparison operator <= used instead of >. This seems like a copy-paste error while trying to convert the previous `require` expression to a `revert` with custom error.

Impact: Borrow will never work for BigBang markets.

Likelihood: High (very likely that the protocol will enforce `totalBorrowCap`) + Impact: High (failure of key operation which initiates the Tapioca protocol) = Severity: Critical

Recommendation: Change the comparison operator from <= to >:

```
  if (totalBorrowCap > 0) {
- if (totalBorrow.elastic <= totalBorrowCap)
+ if (totalBorrow.elastic > totalBorrowCap)
      revert BorrowCapReached();
```

Tapioca-DAO: Fixed in PR 270.

5.2 High Risk

5.2.1 `TWAML` logic is vulnerable to first depositors manipulations

Severity: High Risk

Context: TapiocaOptionBroker.sol#L312-L342 twTAP.sol#L313-L344

Description: Due to the relative logic, which requires 10 BP and up to 10% of the total deposit to influence values of the `Pool.cumulative`.

It is possible for the first few attackers to supply very low values, as a means to raise the `Pool.cumulative` to an inflated value that would allow their small deposits to receive extremely high multiples. They would then Perform as High a deposit as they can, which would force future depositor into locking their tokens for hundreds of years in order to achieve similar multipliers.

Proof of concept:

The proof of concept is based on the Brute Force Script from the issue "Expiry overflow allows attacker to claim majority of rewards and gain voting power while having a liquid lock".

- With `1e18` as start:

```
amt 86449636655527339028
duration 73906576998535027
additionalMultipler 7390657699853502700000
totalAmtNeeded 86536086292182866367209
```

- With `1000_e18`:

```
totalAmtNeeded 100052748793368553625189
lastGoodDuration 6548483419265589000
twtap.spent 100052748793368553625189
totalMultiplier 597125930706501781700000
```

- With 10% Weight Factor: The attack is not substantially mitigated as 10% of close to zero is still a marginally low amount

```
lastGoodDuration 24798408388000
twtap.spent 106718957163359378642382
totalMultiplier 43501075479000000
```

Now that we demonstrated the possibility of achieving incredibly high multipliers, let's show that the first few depositors also have the ability of dragging the cumulative down to close to zero. The proof of concept is based on the same original Mock Test, with an updated Test:

```
function testToZero() public {
    TwTap twtap = new TwTap();

    // Reverse binary search on duration
    uint256 lastGoodDuration = twtap.EPOCH_DURATION() * 4;

    uint256 totalMultiplier;

    uint256 amt;
    uint256 duration;

    uint256 totalAmtNeeded;

    // Normal Lock
    twtap.participate(lastGoodDuration, 10_0000);
    console2.log("cumulative start", twtap.cumulative());

    // Smaller lock, proof we can drag down
    twtap.participate(lastGoodDuration / 2, 10_0000);
    console2.log("cumulative start", twtap.cumulative());

    // Drag down to theoretical minmum
    while(twtap.cumulative() > 604557) {
        twtap.participate(twtap.EPOCH_DURATION() + 1, 10_0000);
        console2.log("cumulative start", twtap.cumulative());
    }
}
```

By locking with smaller durations, we are able to drag down the cumulative to the theoretical minimum which is around the seconds in a week (since we cannot lock for less than a week + 1 second). Running the above yields:

```
Logs:
    cumulative start 2493654
    cumulative start 1410284
    cumulative start 1007756
    cumulative start 865236
    cumulative start 798648
    cumulative start 753690
    cumulative start 716888
    cumulative start 684658
    cumulative start 655647
    cumulative start 629111
    cumulative start 604557
```

This demonstrates that we can drag down cumulative, allowing us to complete the proof of concept with the following steps:

- Raise Cumulative via small locks of maximum duration (proof of concept 1).
- Lock as much as we're willing to lose with incredibly long duration to achieve a super majority of the multiplier.
- Bring the cumulative down with dust locks.

- Lock a bigger amount to make it more expensive / impossible for anybody else to perform this (requires Whale level of Tap or Lp token).

Recommendation: Cap the max lock duration and add a fixed minimum lock size as to reduce the area for economic manipulation.

5.2.2 It is possible to exercise TAP option an extra time compared to lock duration

Severity: High Risk

Context: TapiocaOptionBroker.sol#L506-L512, TapiocaOptionBroker.sol#L350-L360

Description: Due to mix of the epoch based and block time based logic it is still possible to execute oTAP position an extra time compared to the actual lock time, i.e. being locked by `k * EPOCH_DURATION + epsilon` time, where `epsilon` is small, receive option execution payoffs `k + 1` times. This is a C4-189 follow-up.

Proof of concept:

1. In the end of epoch `1` call `lock()` with `lockDuration = EPOCH_DURATION + epsilon` and then `participate()`.
2. In the end of epoch `2` when `block.timestamp > tOLPLockPosition.lockTime + EPOCH_DURATION` call `exerciseOption()` and receive first option.
3. after `newEpoch()` triggering epoch `3` was run and some time has passed so it is `block.timestamp >= lockPosition.lockTime + EPOCH_DURATION + epsilon` call `exerciseOption()`, receive second option, and close the lock with `exitPosition()` and `unlock()`.

`_isPositionActive()` in (2) and (3) will be true as `epoch == 3 = _timestampToWeek(_lock.lockTime + _lock.lockDuration)` (see TapiocaOptionBroker.sol#L620-L631):

```
function _isPositionActive(
    LockPosition memory _lock
) internal view returns (bool isPositionActive) {
    if (_lock.lockTime <= 0) revert PositionNotValid();
    if (_isSGLInRescueMode(_lock)) revert SingularityInRescueMode();
    // see the line below
    uint256 expiryWeek = _timestampToWeek(
        _lock.lockTime + _lock.lockDuration
    );

    isPositionActive = epoch <= expiryWeek;
}
```

also TapiocaOptionBroker.sol#L599-L603:

```
function _timestampToWeek(
    uint256 timestamp
) internal view returns (uint256) {
    return ((timestamp - emissionsStartTime) / EPOCH_DURATION);
}
```

Accounting wise they will write themselves in both epoch `2` and `3` of `netDepositedForEpoch` as first entry is `epoch` based, which is equal to `1` on entry, while the last entry is lock duration based (see TapiocaOptionBroker.sol#L350-L360):

```
// Record amount for next epoch exercise
// see below
netDepositedForEpoch[epoch + 1][lock.sglAssetID] += int256(
    uint256(lock.ybShares)
);
// see below
uint256 lastEpoch = _timestampToWeek(lock.lockTime + lock.lockDuration);
// And remove it from last epoch
// Math is safe, check `_emitToGauges()`
netDepositedForEpoch[lastEpoch + 1][lock.sglAssetID] -= int256(
    uint256(lock.ybShares)
);
```

I.e. the rewards will be properly diluted, but attacker's locking time will not be corresponding to the payoff.

How competitive `newEpoch()` execution was doesn't look to matter here: if other participants run it quickly, the attack is still possible; if not, the attacker can run it themselves and proceed.

Impact: attacker has removed one epoch of rewards from the long term stakers, receiving `2` tapOFT payoffs for `1` epoch long staking. More generally, an attacker can add `1` epoch of option rewards in excess to their actual locking time (as `epsilon` can be made minutes long and not significant position locking wise).

This is a violation of base protocol token economy:

> Lenders with active oTAP positions will receive oTAP shares from the DSO program every week that their position remains locked, proportional to their positions share of the total supplied locked liquidity in the respective market

Likelihood: High (no low probability prerequisites) + Impact: Medium (`1` epoch of rewards is being stolen from long-term lockers) = Severity: High.

Recommendation: Consider enforcing `lockPosition.lockDuration` to be `EPOCH_DURATION` multiplier. There looks to be little downside in this, while tokenomics wise there is no dependency on allowing the whole set of arbitrary locking durations.

Tapioca: Fixed in PR 155.

Spearbit: Fix looks ok.

5.2.3 Expiry overflow allows attacker to claim majority of rewards and gain voting power while having a liquid lock

Severity: High Risk

Context: twTAP.sol#L350-L351

Description: The following report demonstrates how an attacker could fairly cheaply:

- Massively inflate the value of their locked Tap token (by locking small amounts for long durations).
- Use an overflow to instantly break their own Tap deposits while still being eligible for rewards.

The report is based on the following proof of concept which demonstrates that it is possible to cause an overflow on `expiry`.

Seems like a concern if I'm understanding this right but I may not be following any simplifying assumptions made for the proof of concept. So the impact is perpetual rewards + voting power which seems serious enough for a High severity (High+Med=High).

A similar issue was found in the C4 contest, with a reduced impact:

- Ability to steal all rewards.
- Ability to inflate voting power to arbitrary value.

- Close to zero cost requirement as the locks can be instantly unlocked due to the overflow.

`twTap` allows for any arbirarily long duration (see twTAP.sol#L287-L294):

```
/// @param _duration The duration of the lock
function participate(
    address _participant,
    uint256 _amount,
    uint256 _duration
) external whenNotPaused nonReentrant returns (uint256 tokenId) {
    if (_duration < EPOCH_DURATION) revert LockNotAWeek(); /// @audit Can we get rewards for epoch + 2?
↪  While being locked for less than epoch + 2 but more than epoch + 1?
```

It's "line of defense" is this line (see twTAP.sol#L303-L304):

```
if (magnitude >= pool.cumulative * 4) revert NotValid(); /// @audit Can this be used to dos? |
↪  pool.cumulative is 0 no? = This always reverts
```

The magnitude is influenced by the amount of voting power coming from previous locked TAP that had voting power.

Because this is a relative (10 BPS) comparison, by repeatedly locking a small amount of TAP, we create a scenario in which we can make `pool.cumulative` growth, at little cost. By doing that, we're able to "unlock" the ability to set `duration` to an extremely long amount. This value will:

- Give us an extra-ordinary percentage of the rewards, while locking marginal amounts.
- Duration will overflow, effectively making the locks broken the second they are created, while still entitling us to the rewards.

Proof of concept: Through (unrefined) brute force I'm able to determine the theoretical maximum duration before a revert. This is done by always locking 10 BPS up to `100_000e18`, and using simple binary division to brute force some value for which the next lock of the lowest amount but highest magnitude is possible.

The result is: `3157671059092460546000` seconds. `log_2(1705653936+3157671059092460546000)` is `71`, meaning that the expiry can be made to overflow (since it relies on a `uint56`.

Result:

```
additionalMultipler 3157671059092460546000000
lastGoodDuration 3157671059092460546000
twtap.spent 100083770599685131055054
totalMultiplier 294766236865384733173590000
```

Brute force script:

```
// SPDX-License Identifier: MIT

pragma solidity 0.8.17;

import "forge-std/Test.sol";
import "forge-std/console2.sol";


contract TwTap {

    uint256 public EPOCH_DURATION = 7 days;
    uint256 public constant MIN_WEIGHT_FACTOR = 10; // In BPS, 0.1%

    uint256 public cumulative = EPOCH_DURATION;
    uint256 public totalDeposited = 0;
    uint256 public averageMagnitude = 0;

    uint256 public totalParticipants;
```

```
    uint256 public spent = 0;

    function participate(uint256 duration, uint256 amount) external returns (uint256) {
        require(duration > EPOCH_DURATION, "LockNotAWeek");

        // Transfer TAP to this contract
        spent += amount;

        uint256 magnitude = computeMagnitude(duration, cumulative); // This is just duration and prev
        // Revert if the lock 4x the cumulative||| But the impact of locking different weight should be
↪   counted in some way
        require(magnitude < cumulative * 4, "Magnitude too big");
        uint256 multiplier = computeTarget( // magnitude * dMax / cumulative | clamp(dMAX, dMin)
            1_000_000,
            100_000,
            magnitude, /// NOTE: Basically based on duration
            cumulative
        );

        // Calculate twAML voting weight
        bool divergenceForce;
        bool hasVotingPower = amount >=
            computeMinWeight(totalDeposited, MIN_WEIGHT_FACTOR);
        if (hasVotingPower) { /// @audit Not idempotent, ordering matters
            totalParticipants++; // Save participation
            averageMagnitude =
                (averageMagnitude + magnitude) /
                totalParticipants; // compute new average magnitude | // new Magnitude / total? ///
↪   @audit This is NOT average, looks OFF

            // Compute and save new cumulative
            divergenceForce = duration >= cumulative; /// if duration > SUM(prev_durations)

            if (divergenceForce) {
                cumulative += averageMagnitude;
            } else {
                // TODO: Strongly suspect this is never less. Prove it.
                if (cumulative > averageMagnitude) {
                    cumulative -= averageMagnitude;
                } else {
                    cumulative = 0;
                }
            }

            // Save new weight
            totalDeposited += amount;
        }

        return duration * multiplier;
    }

    function getMinWeight() external view returns (uint256) {
        return computeMinWeight(totalDeposited, MIN_WEIGHT_FACTOR);
    }

    function computeMinWeight(
        uint256 _totalWeight,
        uint256 _minWeightFactor
    ) internal pure returns (uint256) {
        uint256 mul = (_totalWeight * _minWeightFactor);
```

```
        return mul >= 1e4 ? mul / 1e4 : _totalWeight; /// @audit First few times this can be zero, if a
↪   small amount is locked
    }

    function computeMagnitude(
        uint256 _timeWeight,
        uint256 _cumulative
    ) internal pure returns (uint256) {
        return /// @audit Safe from overflow by definition sqrt(cum * cum) == cum
            sqrt(_timeWeight * _timeWeight + _cumulative * _cumulative) -
            _cumulative;
    }

    function computeTarget(
        uint256 _dMin,
        uint256 _dMax,
        uint256 _magnitude,
        uint256 _cumulative
    ) internal pure returns (uint256) {
        if (_cumulative == 0) {
            return _dMax;
        }
        uint256 target = (_magnitude * _dMax) / _cumulative; /// @audit if magnituded / cum >= 1 -> dMax
        target = target > _dMax ? _dMax : target < _dMin ? _dMin : target;
        return target;
    }

    // babylonian method
↪   (https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylonian_method)
    function sqrt(uint256 y) internal pure returns (uint256 z) {
        if (y > 3) {
            z = y;
            uint256 x = y / 2 + 1;
            while (x < z) {
                z = x;
                x = (y / x + x) / 2;
            }
        } else if (y != 0) {
            z = 1;
        }
    }
}

contract ExampleTest is Test {
    function testTheTwap() public {
        TwTap twtap = new TwTap();

        // Reverse binary search on duration
        uint256 lastGoodDuration = twtap.EPOCH_DURATION() * 4;

        uint256 totalMultiplier;

        uint256 amt;
        uint256 duration;

        // Amt is always the min required
        while(twtap.spent() < 100_000e18) {
            // duration reverse loop
            amt = twtap.getMinWeight() > 0 ? twtap.getMinWeight() : 1;
            bool success;
            duration = lastGoodDuration;
```

```
            while(!success) {
                try twtap.participate(duration, amt) returns (uint256 additionalMultipler) {
                    lastGoodDuration = duration * 1000; // Increase by 1k so we have a chance at
↪  expanding next loop
                    success = true; // Done with this iteration
                    totalMultiplier += additionalMultipler;
                    console2.log("additionalMultipler", additionalMultipler);
                } catch {
                    duration = duration / 2; // Cut by half each time
                }
            }
        }

        console2.log("lastGoodDuration", lastGoodDuration);
        console2.log("twtap.spent", twtap.spent());
        console2.log("totalMultiplier", totalMultiplier);
    }
}
```

Weaponizing the Overflow: Through the above, we demonstrated that an overflow is possible. To weaponize it we would:

- Find optimal amount at which locking 1 wei (or more) of TAP allows us to maximize the gains from the inflated duration.
- Lock, permanently influencing future rewards (as `w0` and `w1` do not overflow).
- Be able to unlock at any time, as expiry has oveflown, meaning it will be `< block.timestamp`.

Refining the proof of concept: We can further refine the brute force by changing the divisor from 2 to a smaller fraction (e.g. `X * 4 / 5`). We can find the cost of the attack by simply summing up the sum of the minAmounts, in the case of an initial deposit the cost is `4.9e-11`:


```
duration 73906576998535027
additionalMultipler 7390657699853502700000
totalAmtNeeded 49324178
```

Updated proof of concept:

```
function testTheTwap() public {
    TwTap twtap = new TwTap();

    // Reverse binary search on duration
    uint256 lastGoodDuration = twtap.EPOCH_DURATION() * 4;

    uint256 totalMultiplier;

    uint256 amt;
    uint256 duration;

    uint256 totalAmtNeeded;

    // Amt is always the min required
    while(twtap.spent() < 100_000e18) {
        // duration reverse loop
        amt = twtap.getMinWeight() > 0 ? twtap.getMinWeight() : 1;
        totalAmtNeeded += amt;
        bool success;
        duration = lastGoodDuration;

        while(!success) {
            try twtap.participate(duration, amt) returns (uint256 additionalMultipler) {
```

```
                lastGoodDuration = duration * 1000; // Increase by 1k so we have a chance at expanding
↪  next loop
                success = true; // Done with this iteration
                totalMultiplier += additionalMultipler;
                console2.log("amt", amt);
                console2.log("duration", duration);
                console2.log("additionalMultipler", additionalMultipler);
                console2.log("totalAmtNeeded", totalAmtNeeded);
                assert(duration < type(uint56).max);
            } catch {
                duration = duration / 2;
            }
        }
    }

    console2.log("lastGoodDuration", lastGoodDuration);
    console2.log("twtap.spent", twtap.spent());
    console2.log("totalMultiplier", totalMultiplier);
}
```

We can edit the updated proof of concept with an initial locked TAP to show that the cost of the attack is roughly 100 times the initial deposit.

Adapting for initial deposit: By changing `start` where start is `amt = twtap.getMinWeight() > 0 ? twtap.getMinWeight() : START;`

- With 1e18 as Start:

```
amt 86449636655527339028
duration 73906576998535027
additionalMultipler 7390657699853502700000
totalAmtNeeded 865360862921828663672O9
```

- With 1000_e18 as Start:

```
totalAmtNeeded 1000527487933685536251 89
lastGoodDuration 6548483419265589000
twtap.spent 100052748793368553625189
totalMultiplier 5971259307065017817000 00
```

Recommendation: Cap the maximum duration to a rational value (e.g. 4 years). An exploit such as this could be found via invariant testing by:

- Setting all handlers.
- Checking as test that no token can have a duration that is above X.
- Checking as test that no token can have expiry that is before now (meaning it's unlocked the second it's created).

Tapioca: Acknowledged. I believe this is fixed by the issue "`TWAML` logic is vulnerable to first depositors manipulations".

Spearbit: Acknowledged.

5.2.4 Missing dust removal and conversion of decimals on `removeAndRepayData` amount fields in `USDOMarketModule.removeAsset()` may lead to potential loss/inflation of user funds

Severity: High Risk

Context: USDOMarketModule.sol#L29-L80

Description: Unlike all other cross-chain operations, `USDOMarketModule.removeAsset()` misses dust removal with `_removeDust()` and conversion of decimals with `_ld2sd()` for `removeAndRepayData` amount fields of `removeAmount`, `repayAmount` and `collateralAmount` on the source chain and the corresponding reverse conversions on the destination chain.

Impact: This will lead to cross-chain accounting mismatches at the very least and potential loss/inflation of user funds due to any conversion difference in underlying asset decimals between source and destination chains.

Likelihood: Medium (Depends on difference in underlying asset decimals between source and destination chains) + Impact: High (Assuming loss/inflation of user funds due to any difference in underlying asset decimals) = Severity: High.

Recommendation: Consider applying dust removal and conversion of decimals on all `removeAndRepayData` amount fields in `USDOMarketModule.removeAsset()` on the source chain and the corresponding reverse conversions on the destination chain.

Tapioca: Acknowledged. We're applying _toLD transformation on V2 migration already.

Spearbit: Acknowledged.

5.2.5 Incorrect refund may cause loss of sender's cross-chain `sendAndLendOrRepay()` deposit amount on the destination chain

Severity: High Risk

Context: USDOMarketModule.sol#L123-L128, USDOMarketModule.sol#L134-L159, USDOMarketDestinationModule.sol#L81-L87

Description: The cross-chain USDO `sendAndLendOrRepay()` flow takes `_from` and `_to` parameters where USDO is debited at sender's `_from` address on the source chain with `_to` being the receiver beneficiary of the destination chain operations. It appears to be an assumption that even if the operation fails on the destination chain, the refund beneficiary there should be `_to` or that `_to == _from`. While this may be the intended but unspecified behavior, a reasonable expectation would be that if this flow fails on the destination chain for any reason, the refund there should go back to the sender's `_from` address that was debited on the source chain, because it could be different from the receiver `_to` and `_to` may only be intended as a receiver beneficiary of successful lend/repay operations.

However, `USDOMarketModule.sendAndLendOrRepay()` does not encode and send over `_from` address in `_lzSend()` and `lendParams.depositAmount` is instead refunded to the `to` address if this operation fails on the destination chain for any reason.

Impact: Incorrect refund may cause loss of sender's cross-chain `sendAndLendOrRepay()` amount on the destination chain.

Likelihood: Medium (requires `_to != _from` and `lendInternal()` failure on destination chain) + Impact: High (Loss of transaction amount) = Severity: High

Recommendation: Consider passing `_from` as part of `lzPayload` in the `_lzSend()` of `USDOMarketModule.sendAndLendOrRepay()` and transfer `lendParams.depositAmount` to that address on failure at destination chain.

Tapioca: Acknowledged. Not valid anymore with V2.

Spearbit: Acknowledged.

5.2.6 All cross-chain USDO and TOFT flows using approvals may be susceptible to permit-based DoS griefing

Severity: High Risk

Context: MarketERC20.sol#L207-L317, USDOCommon.sol#L20-L151, TOFTCommon.sol#L18-L148, USDOGenericModule.sol, USDOMarketDestinationModule.sol, USDOOptionsDestinationModule.sol, BaseTOFTGenericModule.sol, BaseTOFTMarketDestinationModule.sol, BaseTOFTOptionsDestinationModule.sol, Trust Security DoS Bug Bounty Report

Description: Trust Security recently published a report on their bug bounty disclosure of a permit-based DoS griefing exploit vector which apparently affected 30+ projects. This issue affects cross-chain USDO and TOFT flows which use permit-based approvals.

As described in the report, permit-based DoS griefing is possible because of the following reasons:

1. EIP-2612 defines the ERC20 extension "Permit" which allows users to sign an allowance approval offchain instead of calling `approve()`.
2. Given that `msg.sender` of `permit()` is ignored, anyone can forward it by design, but this also means that they can be front-run by others extracting signature parameters from the observed transaction.
3. The use of nonces prevent permits to be replayed which means that if anyone front-runs a permit, the original permit will revert.
4. Reverting permits may be acceptable if they are standalone, but if they are part of a broader logic then that entire flow reverts.
5. For scenarios where there are fallback options/paths, such a griefing DoS is short-term, but otherwise the DoS is long-term.

Several USDO and TOFT market/option/other flows use bundled permit-based approvals and revokes which are executed on the destination chain via calls to `_callApproval()`. Attackers may extract signature data from the transactions and front-run such flows using standalone approval+revoke to consume the original approval+revoke nonces thereby causing the entire flows to revert in their calls thereafter due to insufficient allowances.

Impact: Because both approve and revoke can be front-run, allowances could be reset back to 0 thus causing all cross-chain USDO and TOFT flows using approvals to be susceptible to reverts via permit-based DoS where an attacker could grief by repeatedly front-running such permit-dependent flows on destination chain even during any retries.

Likelihood: High (Attacker can always capture-frontrun approvals and revokes) + Impact: Medium (Griefing DoS across critical cross-chain flows) = Severity: High.

Recommendation: Consider revisiting/redesigning the bundled permit-based cross-chain flows.

Tapioca: Confirmed that the v2 migration solves this problem by having approvals being run on a separated, isolated transaction.

5.2.7 Second phase of `aoTAP` distributes different amounts with greater discounts than specified in documentation

Severity: High Risk

Context: AirdropBroker.sol#L91-L94, Tapioca Option Airdrop phase-two-core-tapioca-guild

Description: Tapioca Option Airdrop documentation specifies the below for second phase distribution:

> 500,000 TAP will be allocated to the Tapioca Guild in the form of aoTAP call options with a 48-hour expiry.
>
> 1. OG Pearls - 45 Members (50% Discount) - 9000 oTAP (~200 oTAP each)
> 2. Tapiocans - 416 Members (40% Discount) - 85,000 oTAP (~200 oTAP each)
> 3. Oysters - 1870 Members (33% Discount) - 336,000 oTAP (~190 oTAP each)

4. Sushi Frens - 365 Members (25% Discount) - 70,000 oTAP (~190 oTAP each)

whereas, the implementation has:

```
// [OG Pearls, Tapiocans, Oysters, Cassava]
bytes32[4] public phase2MerkleRoots; // merkle root of phase 2 airdrop
uint8[4] public PHASE_2_AMOUNT_PER_USER = [200, 190, 200, 190];
uint8[4] public PHASE_2_DISCOUNT_PER_USER = [50, 40, 40, 33];
```

From above, it appears that the implementation of second phase of `oTAP` distributes 190 `oTAP` to Tapiocans and 200 `aoTAP` to Oysters instead of the other way around. It also appears to give a greater discount of 40% instead of 33% to Oysters and 33% instead of 25% to Sushi Frens.

Impact: Second phase of `aOTAP` distributes different amounts with greater discounts than specified in documentation. The greater than intended discount leads to loss of funds for the protocol treasury and may also affect the fairness of the airdrop from the community's perspective.

Likelihood: High + Impact: Medium (loss of funds to Tapiocans and Protocol) = Severity: High.

Recommendation: Fix the implementation or correct the documentation to bring them in sync to clarify this mismatch.

Tapioca: Addressed in PR 149.

5.2.8 Incorrect refund address causes loss of cross-chain `lockTwTapPosition()` amount to users

Severity: High Risk

Context: BaseTapOFT.sol#L101-L118, BaseTapOFT.sol#L122, BaseTapOFT.sol#L153-L156, BaseTapOFT.sol#L165-L172

Description: The cross-chain `lockTwTapPosition()` operation takes a `to` argument as "*The address to add the twTAP position to*". However, it debits TapOFT from `msg.sender`, which may be different from `to`. While this may be the intended but unspecified behavior, a reasonable expectation would be that if this operation fails on the destination chain for any reason, the refund should go back to the same sender address that was debited on the source chain.

While `lockTwTapPosition()` encodes `msg.sender` in the `lzPayload`, that address is ignored while decoding on the destination chain. Upon failure of `twTap.participate` for some reason, the amount is refunded to `to` address.

Impact: Incorrect refund address causes loss of cross-chain `lockTwTapPosition()` amount to users when `msg.sender` is different from `to` address and `twTap.participate` fails.

Likelihood: Medium (requires `msg.sender != to` and `twTap.participate` failure on destination chain) + Impact: High (loss of TapOFT amount) = Severity: High.

This is similar to the issue "Incorrect refund address causes loss of cross-chain `sendForLeverage()` amount to users".

Recommendation: Do not ignore the `msg.sender` on decoding and use that address instead of `to` in `_transferFrom()` for refund. Alternatively, reconsider if `msg.sender` on the source chain should really be treated as an approved/operator address and the TapOFT should really be debited from `to` instead of `msg.sender` after an allowance check.

Tapioca: Addressed in PR 139.

5.2.9 SGL liquidation will generally miss the fees and will not update `totalAsset.elastic` resulting in asset freeze

Severity: High Risk

Context: SGLLiquidation.sol#L124-L156, SGLLiquidation.sol#L251-L262

Description: SGLLiquidation's `_liquidateUser()` implies that `_liquidatorReceiver` will top up the YieldBox balance of the contract, but due to `_swapCollateralWithAsset()` logic (which is the same as in BBLiquidation), direct asset token transfer is the only allowed path instead (see SGLLiquidation.sol#L124-L156):

```
function _swapCollateralWithAsset(
    uint256 _collateralShare,
    IMarketLiquidatorReceiver _liquidatorReceiver,
    bytes memory _liquidatorReceiverData
) private returns (uint256 returnedShare, uint256 returnedAmount) {
    // ...
    //msg.sender should be validated against `initiator` on IMarketLiquidatorReceiver
    _liquidatorReceiver.onCollateralReceiver(
        msg.sender,
        address(collateral),
        address(asset),
        collateralAmount,
        _liquidatorReceiverData
    );
    uint256 assetBalanceAfter = asset.balanceOf(address(this));
    // see the line below
    returnedAmount = assetBalanceAfter - assetBalanceBefore;
    if (returnedAmount == 0) revert OnCollateralReceiverFailed();
    // see the line below
    returnedShare = yieldBox.toShare(assetId, returnedAmount, false);
}
```

As without it the `_liquidateUser()` logic will revert (see SGLLiquidation.sol#L325-L331):

```
(uint256 returnedShare, ) = _swapCollateralWithAsset(
    collateralShare,
    _liquidatorReceiver,
    _liquidatorReceiverData
);
// see the line below
if (returnedShare < borrowShare) revert AmountNotValid();
```

This way most of the times it will be `extraShare == callerShare == feeShare == 0` as neither `totalAsset.elastic` nor `yieldBox.balanceOf(address(this), assetId)` are modified yet, so generally there will be no difference between the two (see SGLLiquidation.sol#L251-L262):

```
function _extractLiquidationFees(
    uint256 borrowShare,
    uint256 callerReward
) private returns (uint256 feeShare, uint256 callerShare) {
    // see the line below
    uint256 returnedShare = yieldBox.balanceOf(address(this), assetId) -
        uint256(totalAsset.elastic);
    uint256 extraShare = returnedShare > borrowShare
        ? returnedShare - borrowShare
        : 0;

    callerShare = (extraShare * callerReward) / FEE_PRECISION; //  y%  of profit goes to caller.
    feeShare = extraShare - callerShare; // rest goes to the fee
```

So in general no fees will be recorded on liquidations. More importantly, since `returnedShare == feeShare ==`

`callerShare == 0` the `totalAsset.elastic` will not be updated (see SGLLiquidation.sol#L285):

```
totalAsset.elastic += uint128(returnedShare - feeShare - callerShare);
```

This will hide the `returnedAmount` assets above from the depositors, making them frozen with the contract. As `skim` option operates with YB balance (so it can't reach the funds that weren't deposited in the first place) and there looks to be no other mechanics to retrieve unaccounted assets (see SGLCommon.sol#L194-L208):

```
function _addTokens(
    address from,
    address,
    uint256 _assetId,
    uint256 share,
    uint256 total,
    bool skim
) internal {
    if (skim) {
        if (share > yieldBox.balanceOf(address(this), _assetId) - total)
            revert TooMuch();
    } else {
        yieldBox.transfer(from, address(this), _assetId, share);
    }
}
```

Impact: liquidation obtained `returnedAmount` will be frozen with SGL and lost for the depositors.

Likelihood: High + Impact: Medium = Severity: High.

Recommendation: Consider either adding the deposit of the `returnedShare` back to YieldBox in the end of SGL's `_swapCollateralWithAsset()` (and `if (callerShare > 0) {yieldBox.depositAsset(); ...}` and same fee transfer logic will need to be rewritten to be internal YB transfer instead in `_extractLiquidationFees()`).

Tapioca: Fixed in PR 307.

Spearbit: Fix looks ok.

5.2.10 Liquidation repaid debt isn't reinvested in YB, while system accounting assumes that it is

Severity: High Risk

Context: SGLLiquidation.sol#L264-L295

Description: Liquidation repaid debt to be instantly invested in YB because the system assumes that all assets that are not lent out behave just like YB strategy. Instead now the `asset` received from `_liquidatorReceiver` are being left on the contract balance. These funds do not yield along with the corresponding YB strategy.

Even if this investing is carried out manually or periodically by the keeper there will be an accounting mismatch due to gap in the actions (i.e. this additional action should move `totalAsset.elastic` accordingly, but there is no such code at the moment). Any subsequent `yieldBox.toAmount(assetId, _totalAsset.elastic, false)` kind of logic will produce biased estimate of the system assets as collateral proceeds were in fact left residing on the balance, not behaving like YB strategy. Say `yieldBox.toAmount` will produce `101` because YB has yielded `1%`, but it will be in fact `100.5` assets as some are residing on the balance, not yielding anything.

Impact: system accounting mismatch. It can lead to system insolvency as withdrawing depositors are returned with funds as if they were all invested in YB. If some yield is in fact missed instead the last depositors will not be able to withdraw.

Likelihood: Medium + Impact: High = Severity: High.

Recommendation: Consider automatically depositing the funds received on liquidation to YB.

Tapioca: Fixed in PR 307.

Spearbit: Fix looks ok.

5.2.11 Limiting the wrapping of MetaTOFTs only to the host chain breaks a core protocol invariant

Severity: High Risk

Context: Tapioca MetaTOFTs, mTapiocaOFT.sol#L107-L118, BaseTOFT.sol#L77-L80, mTapiocaOFT.sol#L123-L127, mTapiocaOFT.sol#L135-L145

Description: While the wrapping and unwrapping of Tapioca TOFTs are limited to the host chain, Tapioca MetaTOFTs are documented as below:

> "mTOFT" or "Meta-tOFT" are an extension of the tOFT which act as a "liquidity reunification wrapper", for assets deployed on multiple blockchains such as with mtWSTETH or mtETH. Meta tOFT offers the ability for users to deposit Lido wstETH from Optimism or Arbitrum into one market, or for users to deposit WETH from Ethereum, Optimism, and Arbitrum in one market via mtWETH.

The protocol is supposed to implement MetaTOFTs by allowing their wrapping and unwrapping on all "connected chains", which is a core protocol invariant. The chain with `_hostChainID` is enabled as one of the `connectedChains` in the constructor and the owner later updates any other chain's `connectedChains` whitelist status using `updateConnectedChain()`. While the current implementation allows their unwrapping on all connected chains, it incorrectly limits their wrapping to only the host chain thereby breaking the invariant.

Impact: Limiting the wrapping of MetaTOFTs only to the host chain breaks a core protocol invariant.

Likelihood: High + Impact: Medium (forces users to wrap only on the host chain) = Severity: High.

Recommendation: Allow wrapping of MetaTOFTs on all connected chains similar to their unwrapping. For example, consider removing `onlyHostChain` modifier on `mTapiocaOFT.wrap()` and instead add `if (!connectedChains[block.chainid]) revert NotConnected();` check within the function.

Tapioca: Created PR 142.

5.2.12 Cross-chain TOFT `sendForLeverage()` slippage check does not work as expected and may fail

Severity: High Risk

Context: BaseTOFTLeverageModule.sol#L57-L58, BaseTOFTStorage.sol#L97-L104,

Description: Cross-chain TOFT `sendForLeverage()` applies a slippage check on the source chain using `amount` and `swapData.amountOutMin` amounts. However, `amount` refers to the TOFT amount that is being debited and credited on the source and destination chains respectively, whereas `swapData.amountOutMin` refers to the amount of `swapData.tokenOut` on the destination chain. `_assureMaxSlippage()` is therefore applying the check on amounts of two different tokens across chains assuming they have the same underlying value, which is very likely not going to be the case.

Impact: Cross-chain TOFT `sendForLeverage()` slippage check does not work as expected on the source chain and may fail.

Likelihood: High (`swapData.tokenOut` is guaranteed to different from the TOFT) + Impact: Medium (check fails and reverts an important cross-chain operation) = Severity: High.

Recommendation: Consider relying on the slippage check only on the destination chain during the swap instead of failing early on the source chain.

Tapioca: Created PR 140.

5.2.13 Incorrect refund address causes loss of cross-chain `sendForLeverage()` amount to users

Severity: High Risk

Context: BaseTOFTLeverageModule.sol#L51-L55, BaseTOFTLeverageModule.sol#L68-L76, BaseTOFTLeverageModule.sol#L82-L106, BaseTOFTLeverageDestinationModule.sol#L53-L74, BaseTOFTLeverageDestinationModule.sol#L96-L107, BaseTOFTLeverageDestinationModule.sol#L122-L125, USDOLeverageModule.sol, USDOLeverageDestinationModule.sol

Description: The cross-chain `sendForLeverage()` operation takes a `leverageFor` argument as the address for which leverage is being performed. If `msg.sender != leverageFor`, `sendForLeverage()` checks for TOFT `allowance(leverageFor, msg.sender)` on the amount being leveraged but unlike other cross-chain operations which treat `msg.sender` as an approved/operator address, it debits TOFT from `msg.sender`. While this may be the intended but unspecified behavior, a reasonable expectation would be that if this operation fails on the destination chain for any reason, the refund should go back to the same sender address that was debited on the source chain.

However, while `sendForLeverage()` encodes `senderBytes` in the `lzPayload`, that address is ignored while decoding on the destination chain. Upon failure of `leverageDownInternal()` for some reason, the amount is refunded to `leverageFor` address.

Impact: Incorrect refund address causes loss of cross-chain `sendForLeverage()` amount to users when sender address is different from `leverageFor` address and `leverageDownInternal()` fails.

Likelihood: Medium (requires `msg.sender != leverageFor` and `leverageDownInternal()` failure on destination chain) + Impact: High (Loss of transaction amount) = Severity: High

A similar issue is present in USDO `sendForLeverage()` flow.

Recommendation: Do not ignore the `senderBytes` on decoding and use that address instead of `leverageFor` in `_storeAndSend()` for refund. Alternatively, reconsider if `msg.sender` on the source chain should really be treated as an approved/operator address and the leveraged amount of TOFT should be debited from `leverageFor` instead of `msg.sender`.

Evaluate the similar USDO `sendForLeverage()` flow and consider the above recommendations.

5.2.14 `USDO.sendForLeverage` will not work as it's incorrectly applying slippage checks

Severity: High Risk

Context: USDOLeverageModule.sol#L40-L41

Description: `sendForLeverage` is meant to:

- Burn USDO on origin chain.
- Mint it on dstChain.
- Swap USDO for another collateral token as a means to leverage up.

The functions look as follows (see USDOLeverageModule.sol#L28-L41):

```
function sendForLeverage(
    uint256 amount,
    address leverageFor,
    IUSDOBase.ILeverageLZData calldata lzData,
    IUSDOBase.ILeverageSwapData calldata swapData,
    IUSDOBase.ILeverageExternalContractsData calldata externalData
) external payable {
    if (leverageFor != msg.sender) {
        if (allowance(leverageFor, msg.sender) < amount)
            revert AllowanceNotValid();
        _spendAllowance(leverageFor, msg.sender, amount);
    }
    if (swapData.tokenOut == address(this)) revert NotValid();
    _assureMaxSlippage(amount, swapData.amountOutMin);
```

Notably, the `swapData.tokenOut` is not USDO (per the customError), meaning that `swapData.amountOutMin` should be an arbitrary value (in lack of an oracle). Due to this, we must conclude that this check (see USDOLeverageModule.sol#L91-L98):

```
function _assureMaxSlippage(
    uint256 amount,
    uint256 minAmount
) internal pure {
    uint256 slippageMinAmount = amount -
        ((SWAP_MAX_SLIPPAGE * amount) / SLIPPAGE_PRECISION);
    if (minAmount < slippageMinAmount) revert AmountTooLow();
}
```

is incorrect, as it's assuming that the `tokenOut` will have the same denomination and value as USDO, but it will most likely won't.

Recommendation: Consider applying the minOut check only in the destination chain and refactor this check to a simple "non-zero" check. If you wish to enforce a `minOut` check on the source chain, you will have to integrate an oracle of some sort, which may introduce further MEV risks.

Tapioca: Replaced with a non-zero check for slippage in commit 312464c9.

Spearbit: Verified.

5.2.15 Exchange rate timestamp is not recorded on liquidations and can be incorrectly recorded during borrowing, repayment and collateral actions as `updateExchangeRate()` can reset it even when the feed isn't active

Severity: High Risk

Context: Market.sol#L382, BBLiquidation.sol#L37, BBLiquidation.sol#L102, SGLLiquidation.sol#L37, SGLLiquidation.sol#L104

Description: `rateTimestamp` variable, that tracks last known time of exchange rate successful fetch, is not recorded when this rate is retrieved on liquidations, while the corresponding exchange rate variable, `exchangeRate`, is modified. This way the last known time, `rateTimestamp`, becomes stale:

- BBLiquidation.sol#L29-L41

```
function liquidateBadDebt(
    address user,
    address receiver,
    IMarketLiquidatorReceiver liquidatorReceiver,
    bytes calldata liquidatorReceiverData
) external onlyOwner {
    (bool updated, uint256 _exchangeRate) = oracle.get(oracleData);
    if (updated && _exchangeRate > 0) {
        // see the line below
        exchangeRate = _exchangeRate; //update cached rate
    } else {
        _exchangeRate = exchangeRate; //use stored rate
    }
    if (_exchangeRate == 0) revert ExchangeRateNotValid();
```

- BBLiquidation.sol#L87-L106

```
function liquidate(
    address[] calldata users,
    uint256[] calldata maxBorrowParts,
    IMarketLiquidatorReceiver[] calldata liquidatorReceivers,
    bytes[] calldata liquidatorReceiverDatas
) external optionNotPaused(PauseType.Liquidation) {
    if (users.length == 0) revert NothingToLiquidate();
    if (users.length != maxBorrowParts.length) revert LengthMismatch();
    if (users.length != liquidatorReceivers.length) revert LengthMismatch();
    if (liquidatorReceiverDatas.length != liquidatorReceivers.length)
        revert LengthMismatch();

    // Oracle can fail but we still need to allow liquidations
    (bool updated, uint256 _exchangeRate) = oracle.get(oracleData);
    if (updated && _exchangeRate > 0) {
        // see the line below
        exchangeRate = _exchangeRate; //update cached rate
    } else {
        _exchangeRate = exchangeRate; //use stored rate
    }
    if (_exchangeRate == 0) revert ExchangeRateNotValid();
```


- SGLLiquidation.sol#L29-L41

```
function liquidateBadDebt(
    address user,
    address receiver,
    IMarketLiquidatorReceiver liquidatorReceiver,
    bytes calldata liquidatorReceiverData
) external onlyOwner {
    (bool updated, uint256 _exchangeRate) = oracle.get(oracleData);
    if (updated && _exchangeRate > 0) {
        // see the line below
        exchangeRate = _exchangeRate; //update cached rate
    } else {
        _exchangeRate = exchangeRate; //use stored rate
    }
    if (_exchangeRate == 0) revert ExchangeRateNotValid();
```

- SGLLiquidation.sol#L89-L108

```
function liquidate(
    address[] calldata users,
    uint256[] calldata maxBorrowParts,
    IMarketLiquidatorReceiver[] calldata liquidatorReceivers,
    bytes[] calldata liquidatorReceiverDatas
) external optionNotPaused(PauseType.Liquidation) {
    if (users.length == 0) revert NothingToLiquidate();
    if (users.length != maxBorrowParts.length) revert LengthMismatch();
    if (users.length != liquidatorReceivers.length) revert LengthMismatch();
    if (liquidatorReceiverDatas.length != liquidatorReceivers.length)
        revert LengthMismatch();

    // Oracle can fail but we still need to allow liquidations
    (bool updated, uint256 _exchangeRate) = oracle.get(oracleData);
    if (updated && _exchangeRate > 0) {
        // see the line below
        exchangeRate = _exchangeRate; //update cached rate
    } else {
        _exchangeRate = exchangeRate; //use stored rate
    }
    if (_exchangeRate == 0) revert ExchangeRateNotValid();
```

Simultaneously, and unrelated to the liquidations, `updateExchangeRate()` resets `rateTimestamp` even when `updated == false`:

- Market.sol#L366-L383

```
function updateExchangeRate() public returns (bool updated, uint256 rate) {
    (updated, rate) = oracle.get(oracleData);

    if (updated) {
        require(rate != 0, "Market: invalid rate");
        exchangeRate = rate;
        emit LogExchangeRate(rate);
    } else {
        require(
            rateTimestamp + rateValidDuration >= block.timestamp,
            "Market: rate too old"
        );
        // Return the old rate if fetching wasn't successful & rate isn't too old
        // see the line below
        rate = exchangeRate;
    }
    // see the line below
    rateTimestamp = block.timestamp;
}
```

`updateExchangeRate()` is called via `solvent` modifier during borrowing, repayment and collateral management operations.

Impact: `rateValidDuration` ends up not being controlled. Due to `rateTimestamp` not being recorded on liquidations `updateExchangeRate()` will start reverting sooner than `rateValidDuration` dictates, making operations unavailable in excess to the desired logic. Due to `rateTimestamp` being recorded in `updateExchangeRate()` on outdated exchange rate usages within `rateValidDuration` this stale rate can be used perpentually for borrowing, repayment and collateral related actions that utilize `updateExchangeRate()`.

The only prerequisite is any staleness or malfunction of the oracle feed, so cumulative probability here is medium. Usage of a substantially stale exchange rate is a high impact.

Likelihood: Medium + Impact: High = Severity: High.

Recommendation: Consider updating the time on liquidations and not updating it when previously recorded value is used in `updateExchangeRate()`, e.g. BBLiquidation.sol#L29-L41:

```
  function liquidateBadDebt(
      // ...
  ) external onlyOwner {
      (bool updated, uint256 _exchangeRate) = oracle.get(oracleData);
      if (updated && _exchangeRate > 0) {
          exchangeRate = _exchangeRate; //update cached rate
+         rateTimestamp = block.timestamp;
      } else {
          _exchangeRate = exchangeRate; //use stored rate
      }
      if (_exchangeRate == 0) revert ExchangeRateNotValid();
```

- BBLiquidation.sol#L87-L106

```
  function liquidate(
      // ...
  ) external optionNotPaused(PauseType.Liquidation) {
      if (users.length == 0) revert NothingToLiquidate();
      if (users.length != maxBorrowParts.length) revert LengthMismatch();
      if (users.length != liquidatorReceivers.length) revert LengthMismatch();
      if (liquidatorReceiverDatas.length != liquidatorReceivers.length)
          revert LengthMismatch();

      // Oracle can fail but we still need to allow liquidations
      (bool updated, uint256 _exchangeRate) = oracle.get(oracleData);
      if (updated && _exchangeRate > 0) {
          exchangeRate = _exchangeRate; //update cached rate
+         rateTimestamp = block.timestamp;
      } else {
          _exchangeRate = exchangeRate; //use stored rate
      }
      if (_exchangeRate == 0) revert ExchangeRateNotValid();
```


- SGLLiquidation.sol#L29-L41

```
  function liquidateBadDebt(
      // ...
  ) external onlyOwner {
      (bool updated, uint256 _exchangeRate) = oracle.get(oracleData);
      if (updated && _exchangeRate > 0) {
          exchangeRate = _exchangeRate; //update cached rate
+         rateTimestamp = block.timestamp;
      } else {
          _exchangeRate = exchangeRate; //use stored rate
      }
      if (_exchangeRate == 0) revert ExchangeRateNotValid();
```

- SGLLiquidation.sol#L89-L108

```
  function liquidate(
      // ...
  ) external optionNotPaused(PauseType.Liquidation) {
      if (users.length == 0) revert NothingToLiquidate();
      if (users.length != maxBorrowParts.length) revert LengthMismatch();
      if (users.length != liquidatorReceivers.length) revert LengthMismatch();
      if (liquidatorReceiverDatas.length != liquidatorReceivers.length)
          revert LengthMismatch();

      // Oracle can fail but we still need to allow liquidations
      (bool updated, uint256 _exchangeRate) = oracle.get(oracleData);
      if (updated && _exchangeRate > 0) {
          exchangeRate = _exchangeRate; //update cached rate
+         rateTimestamp = block.timestamp;
      } else {
          _exchangeRate = exchangeRate; //use stored rate
      }
      if (_exchangeRate == 0) revert ExchangeRateNotValid();
```

- Market.sol#L366-L383

```
  function updateExchangeRate() public returns (bool updated, uint256 rate) {
      (updated, rate) = oracle.get(oracleData);

      if (updated) {
          require(rate != 0, "Market: invalid rate");
          exchangeRate = rate;
+         rateTimestamp = block.timestamp;
          emit LogExchangeRate(rate);
      } else {
          require(
              rateTimestamp + rateValidDuration >= block.timestamp,
              "Market: rate too old"
          );
          // Return the old rate if fetching wasn't successful & rate isn't too old
         rate = exchangeRate;
      }

-     rateTimestamp = block.timestamp;
  }
```

Tapioca: Fixed in PR 310.

Spearbit: Fix looks ok.

5.2.16 ETH market borrow and repay call for the linked BB markets accrual after the state change, allowing for manipulations

Severity: High Risk

Context: BBBorrow.sol#L50, BBBorrow.sol#L90

Description: BB borrow and repay call `penrose.reAccrueBigBangMarkets()` after the corresponding operation has changed the internal state. The call itself was introduced as a mitigation for C4-1561, but it is not efficient in that regard as the manipulation described there specifically aimed at applying the manipulated state to the markets, which is exactly what achieved by the fix, that first changes the state, then applies it downstream. I.e. current implementation still allows for the attack on stale BB markets with flash loaning ETH, moving ETH market and calling accrue for the secondary ones. The latter is now done automatically by the `borrow()` and `repay()`.

Impact: it is still possible to tamper with interest rates of the secondary BB markets by moving the ETH market. The only prerequisite is low enough activity of those markets, so accrue won't be run too frequently because of

that and there will be enough period of time (e.g., a full day, as C4-1561 showcases) for manipulatable interest to be accumulated.

Since the interest can be material the impact is high, while given the prerequisite the likelihood can be best assessed as medium.

Likelihood: Medium + Impact: High = Severity: High.

Recommendation: Consider to accrue the interest across the linked BB markets before internal state of the ETH market is modified, e.g. BBBorrow.sol#L25-L51:

```
  function borrow(
      address from,
      address to,
      uint256 amount
  )
      external
      optionNotPaused(PauseType.Borrow)
      notSelf(to)
      solvent(from, false)
      returns (uint256 part, uint256 share)
  {
      if (amount < debtStartPoint) revert NotEnough();

      if (amount == 0) return (0, 0);
+     penrose.reAccrueBigBangMarkets();
      uint256 feeAmount = _computeVariableOpeningFee(amount);
      uint256 allowanceShare = _computeAllowanceAmountInAsset(
          from,
          exchangeRate,
          amount + feeAmount,
          asset.safeDecimals()
      );
      if (allowanceShare == 0) revert AllowanceNotValid();
      _allowedBorrow(from, allowanceShare);
      (part, share) = _borrow(from, to, amount, feeAmount);

-     penrose.reAccrueBigBangMarkets();
  }
```

Also see BBBorrow.sol#L59-L91:

```
  function repay(
      address from,
      address to,
      bool,
      uint256 part
  )
      external
      optionNotPaused(PauseType.Repay)
      notSelf(to)
      returns (uint256 amount)
  {
      updateExchangeRate();

      _accrue();
+     penrose.reAccrueBigBangMarkets();

      uint256 partInAmount;
      Rebase memory _totalBorrow = totalBorrow;
      (_totalBorrow, partInAmount) = _totalBorrow.sub(part, true);

      uint256 allowanceShare = _computeAllowanceAmountInAsset(
          to,
          exchangeRate,
          partInAmount,
          asset.safeDecimals()
      );

      if (allowanceShare == 0) revert AllowanceNotValid();
      _allowedBorrow(from, allowanceShare);

      amount = _repay(from, to, part);

-     penrose.reAccrueBigBangMarkets();
  }
```


Tapioca: Fixed in PR 305.

Spearbit: Fix looks ok.

5.2.17 New interest protocol fee accounting is incorrectly computed from `totalBorrow.base`, that biases effective fee allocation from the configured `protocolFee`

Severity: High Risk

Context: SGLCommon.sol#L114

Description: New fee taken on interest accrual from the protocol perspective represents new assets attributable to a specific actor (set later via `refreshPenroseFees()`). This logic isn't done properly, being based on the value of `totalBorrow.base`, which is not the total assets of the system base new addition needs to be calculated from (see SGLCommon.sol#L108-L116):

```
//take accrued values into account
fullAssetAmount =
    yieldBox.toAmount(assetId, _totalAsset.elastic, false) +
    _totalBorrow.elastic;

uint256 feeAmount = (extraAmount * protocolFee) / FEE_PRECISION; // % of interest paid goes to fee
// see the line below
feeFraction = (feeAmount * _totalBorrow.base) / fullAssetAmount;
_accrueInfo.feesEarnedFraction += uint128(feeFraction);
_totalAsset.base = _totalAsset.base + uint128(feeFraction);
```

In other words the correct logic is asset base dilution representing the addition of the protocol owned amount of new interest earned, `feeAmount`, which is in current token units. It is basically an addition of extra assets, which need to have the same logic as an outside investment (new deposit equal to fee amount done by fee recipient). That's why `fullAssetAmount` need also be corrected for `feeAmount` as it should represent the state before the operation.

If `_totalAsset.base` corresponds to `fullAssetAmount - feeAmount`, i.e. all the assets except the new yet unaccounted protocol part, then the new accounting unit to be allocated to the protocol to be a proportional part of `feeAmount` in the whole assets as if it was added from outside, similarly to depositing (see SGLCommon.sol#L211-L229):

```
function _addAsset(
    // ...
    uint256 share
) internal returns (uint256 fraction) {
    Rebase memory _totalAsset = totalAsset;
    uint256 totalAssetShare = _totalAsset.elastic;
    uint256 allShare = _totalAsset.elastic +
        yieldBox.toShare(assetId, totalBorrow.elastic, true);
    fraction = allShare == 0
        ? share
        // see the line below
        : (share * _totalAsset.base) / allShare;
    if (_totalAsset.base + uint128(fraction) < 1000) {
        return 0;
    }
    totalAsset = _totalAsset.add(share, fraction);
    // see the line below
    balanceOf[to] += fraction;
```

To complete the description let's note that `_accrueInfo.feesEarnedFraction` is then being correctly operated in the `totalAsset.base` units, being added to fee recipient's base balance (see Singularity.sol#L489-L499):

```
function refreshPenroseFees()
    external
    onlyOwner
    returns (uint256 feeShares)
{
    address _feeTo = address(penrose);
    // withdraw the fees accumulated in `accrueInfo.feesEarnedFraction` to the balance of `feeTo`.
    if (accrueInfo.feesEarnedFraction > 0) {
        _accrue();
        uint256 _feesEarnedFraction = accrueInfo.feesEarnedFraction;
        // see the line below
        balanceOf[_feeTo] += _feesEarnedFraction;
```

Impact: protocol fee is misstated. It will be lower than the `protocolFee` based one as `totalBorrow.base < totalAsset.base`. The miscalculation happens on every interest accrual operation.

Per high likelihood (occurs every time without low probability preconditions) and medium impact (the effective protocol fee percentage is not `protocolFee`) setting the severity to be high.

Recommendation: Consider computing the fee related asset impact off `_totalAsset.base` and full asset amount before the new fee assets were created (see SGLCommon.sol#L114-116):

```
- feeFraction = (feeAmount * _totalBorrow.base) / fullAssetAmount;
+ feeFraction = (feeAmount * _totalAsset.base) / (fullAssetAmount - feeAmount);
  _accrueInfo.feesEarnedFraction += uint128(feeFraction);
  _totalAsset.base = _totalAsset.base + uint128(feeFraction);
```

Tapioca: PR Created (PR 297) and updated (PR 314.

Spearbit: Fix looks ok.

5.2.18 `liquidateBadDebt` is not repaying any of the debt, removing assets from the depositors and driving the system towards insolvency

Severity: High Risk

Context: BBLiquidation.sol#L29-L78 SGLLiquidation.sol#L29-L79

Description: Both `BigBang` and `Singularity` `liquidateBadDebt()` functions will forgive 100% of the `user` debt:

- SGLLiquidation.sol#L62-L70:

```
// everything will be liquidated; set borrow part and collateral share to 0
uint256 borrowAmount;
(totalBorrow, borrowAmount) = totalBorrow.sub(_userBorrowPart, true);
userBorrowPart[user] = 0;
_yieldBoxShares[user][ASSET_SIG] = 0;

totalCollateralShare -= userCollateralShare[user];
userCollateralShare[user] = 0;
_yieldBoxShares[user][COLLATERAL_SIG] = 0;
```

- BBLiquidation.sol#L61-L69

```
uint256 borrowAmount;
(totalBorrow, borrowAmount) = totalBorrow.sub(
    userBorrowPart[user],
    true
);
userBorrowPart[user] = 0;

totalCollateralShare -= userCollateralShare[user];
userCollateralShare[user] = 0;
```

They will then transfer the `asset` to the `receiver` (see BBLiquidation.sol#L77-L78):

```
asset.safeTransfer(receiver, returnedAmount);
```

However, none of those functions will require a repayment of the debt that is being forgiven, this will cause the system to take on a loss for 100% of the debt that was borrowed, instead of a loss for the shortfall of bad debt.

Impact: depositors will incur a 100% current debt loss of the debt being liquidated by `liquidateBadDebt()`. This is not dictated by the system design, where only debt shortfall might be needed to be incurred as a direct loss (i.e. if loan collateral is 99% of the current loan, it is 1% loss, not 100%, as 99% are recoverable). That's a direct loss for the depositors, conditional on the usage of owner called `liquidateBadDebt()`.

Likelihood: Medium + Impact: High = Severity: High.

Recommendation: Since Tapioca intends on repaying the bad debt in these scenarios, consider changing the code to:

- Have the owner repay 100% of the debt.
- Transfer the collateral to the owner.

This will enforce that the DAO is taking the loss in case of bad debt, if such loss takes place, and receives the surplus otherwise.

Also, it is important to increase `totalAsset.elastic` in `SGLLiquidation` logic to reflect this asset addition and to counteract `totalBorrow.elastic` reduction. I.e. total assets need to remain constant in this approach.

5.2.19 Cross-chain Native TOFT rebalance functionality is broken because ETH cannot be received

Severity: High Risk

Context: BaseTOFT.sol#L499-L516

Description: `sgReceive()` is expected to receive TOFT ETH/ERC20 from Balancer via Stargate router as commented: "*needed for Stargate Router to receive funds from Balancer.sol contract*". This would be the underlying ETH or ERC20 token depending on if the TOFT is native or not. However, `sgReceive()` is missing the `payable` keyword which prevents it from receiving ETH.

Impact: Cross-chain Native TOFT rebalance functionality in `mTapiocaOFT` is broken because ETH cannot be received from Balancer.

Likelihood: High + Impact: Medium = Severity: High.

Recommendation: Add `payable` to `sgReceive()`.

Tapioca: Created PR 126.

5.2.20 Repayment protocol fees are computed off the protocol inception base and can become substantially exaggerated

Severity: High Risk

Context: BBLendingCommon.sol#L122-L133

Description: BBLendingCommon's `_repay()` calculates protocol fee amount from the difference between the current amount being repaid, which is in elastic units, and `part` amount, which is in base units, i.e. units as of system inception. `to` might have entered the system long after it started, and despite their total debt being written in base units it is not what they borrowed.

This way the `accruedFees` and the `feeAmount` derived will be more and more bloated along with the system accruing interest (see BBLendingCommon.sol#L122-L133):

```
(totalBorrow, amount) = totalBorrow.sub(part, true);
userBorrowPart[to] -= part;

amountOut = amount;

yieldBox.withdraw(assetId, from, address(this), amount, 0);
// see below
uint256 accruedFees = amount - part;
if (accruedFees > 0) {
    uint256 feeAmount = (accruedFees * protocolFee) / FEE_PRECISION;
    amount -= feeAmount;
}
```

As an example, let's suppose Bob borrowed `110`, `userBorrowPart[to]` was recorded as `100` since the system has `11` elastic per `10` base when they entered. Afterwards, on repay let's say the `elastic/base = 1.2` and they have to return `120`, then the total amount of interest from which the total fee to be calculated is not `120 - 100`, it is `120 -`

110. I.e. base are abstract units not relevant to the fee calculation, what matters here is moved amount difference, the additional interest incurred by Bob, elastic1 - elastic0 = 120 - 110, while 100 is useful for accounting perspective, for tracking Bob and Alice accounts in the same units, while Alice might have used the system long before or after Bob and dealt with different accruals.

Impact: Over time feeAmount = (accruedFees * protocolFee) / FEE_PRECISION will the more and more substantial part of amount and removing both feeAmount and openingFee will become impossible, the repayment will become frozen in a similar manner as in the issue "Base amount repaid is checked against elastic opening fee, which will freeze all repayments once total elastic inflator become big enough". The notable difference is that all the time before that the protocol fees the system will deduce will be exaggerated.

Since the logic is run on each BB repayment, the probability here is high. The impact of retaining too much USDO as a fee is medium as this jeopardizes the base system accounting, while manual burning doesn't look to be feasible and can be complicated from the operational standpoint. Setting the overall severity to be high this way.

Recommendation: Consider calculating fee on the interest accrual stage (e.g. totalNewProtocolFee = (newInterest * protocolFee) / FEE_PRECISION) and accounting it separately.

Tapioca: Affected code was removed as a part of another mitigation. It was removed after implementing the fix corresponding to the issue "BBLendingCommon is not minting fees which breaks an implicit invariant of CDP Systems".

Spearbit: Confirmed.

5.2.21 Base amount repaid is checked against elastic opening fee, which will freeze all repayments once total elastic inflator become big enough

Severity: High Risk

Context: BBLendingCommon.sol#L116-L117

Description: BBLendingCommon's _repay() compares base part being repaid with elastic _computeRepayFee(to, part) value. Once total base / elastic reversed inflator comes below the opening fee the repayments will become frozen. I.e. it will be openingFee >= part all the time because part will be too small just because the system accumulated big enough interest since inception.

This might not even take too long, say as a hypothetical example with fee = maxMintFee / FEE_PRECISION = 1000 / 1e5 = 0.01 and average interest rate being 20% it will be base / elastic = 1 / 114.5 after 26 years. So mere translation of the amount being repaid to the base units will become more substantial than taking the fee off this amount, so this check will become permanently failing.

Impact: repayment functionality will become frozen once system accumulates enough interest.

Per high protocol and corresponding asset freeze impact and medium probability of this happening while the protocol be in use setting the overall severity to be medium.

Recommendation: Consider making the check against the amount that the fee will be deduced from, e.g.BBLendingCommon.sol#L116-L125:

```
  uint256 openingFee = _computeRepayFee(to, part);
- if (openingFee >= part) revert RepayAmountNotValid();

  openingFees[to] -= openingFee;

  uint256 amount;
  (totalBorrow, amount) = totalBorrow.sub(part, true);
+ if (openingFee >= amount) revert RepayAmountNotValid();
   userBorrowPart[to] -= part;

  amountOut = amount;
```

Tapioca: Code removed during another mitigation. openingFees property was removed after implement a fix for one of the previous issues, it doesn't exist anymore on the master branch.

Spearbit: Confirmed, referenced code was removed.

5.2.22 User can manipulate the borrow/repay mechanism to cause loss of `openingFee` for the protocol

Severity: High Risk

Context: BBLendingCommon.sol#L57

Description: During `borrow` the `openingFees` and `userBorrowPart` are accounted for differently than the `repay` mechanism.

- Borrow:

```
openingFees[to]
userBorrowPart[from]
```

- Repay:

```
openingFees[to]`
userBorrowPart[to]`
```

A user can manipulate this difference by delegating the credit to another address and using a different from/to address during `borrow` & `repay`.

Although `borrowPart` is calculated inclusive of the `feeAmount` and would have to repay the respective amount, the `openingFee` is subtracted from the amount here so that it stays in the contract to be withdrawn later. However, using a different `from` address in `borrow` than the `to` address in `repay` one can make sure the `openingFees` never stays in the contract. Thus the protocol loses the opening fee.

Here is the proof of concept that shows the same:


```
describe('skip opening fee POC', () => {
    it.only('can skip opening fee using different from and to', async () => {
        const {
            wethBigBangMarket,
            weth,
            wethAssetId,
            yieldBox,
            deployer,
            bar,
            usdO,
            __wethUsdcPrice,
            timeTravel,
            eoa1,
        } = await loadFixture(register);

        await weth.approve(yieldBox.address, ethers.constants.MaxUint256);
        await yieldBox.setApprovalForAll(wethBigBangMarket.address, true);

        const wethMintVal = ethers.BigNumber.from((1e18).toString()).mul(
            10,
        );
        await weth.freeMint(wethMintVal);
        const valShare = await yieldBox.toShare(
            wethAssetId,
            wethMintVal,
            false,
        );
        await yieldBox.depositAsset(
            wethAssetId,
            deployer.address,
            deployer.address,
            0,
```

```
        valShare,
    );

    await wethBigBangMarket.addCollateral(
        deployer.address,
        deployer.address,
        false,
        0,
        valShare,
    );

    //borrow
    const usdoBorrowVal = wethMintVal
        .mul(74)
        .div(100)
        .mul(__wethUsdcPrice.div((1e18).toString()));

    await wethBigBangMarket
        .connect(deployer)
        .approveBorrow(eoa1.address, valShare);
    await wethBigBangMarket.connect(eoa1).borrow(
        deployer.address,
        eoa1.address,
        usdoBorrowVal,
    );

    let userBorrowPart = await wethBigBangMarket.userBorrowPart(
        deployer.address,
    );
    expect(userBorrowPart.gt(0)).to.be.true;
    let balanceBefore = await usd0.balanceOf(wethBigBangMarket.address);
    console.log("BBmarket usd0 balance before repay:", balanceBefore);
    const usd0Balance = await yieldBox.toAmount(
        await bar.usdoAssetId(),
        await yieldBox.balanceOf(
            eoa1.address,
            await wethBigBangMarket.assetId(),
        ),
        false,
    );
    expect(usd0Balance.gt(0)).to.be.true;
    expect(usd0Balance.eq(usdoBorrowVal)).to.be.true;

    timeTravel(10 * 86400);
    let borrowfeesDeployer = await wethBigBangMarket.openingFees(deployer.address);
    let borrowfeesA = await wethBigBangMarket.openingFees(eoa1.address);
    console.log("UserA borrow fee:", borrowfeesA);
    console.log("UserDeployer borrow fees:",borrowfeesDeployer );

    //repay
    userBorrowPart = await wethBigBangMarket.userBorrowPart(
        deployer.address,
    );

    await expect(
        wethBigBangMarket.repay(
            deployer.address,
            deployer.address,
            false,
            userBorrowPart,
        ),
    ).to.be.reverted;
```

```
        const usd0Extra = ethers.BigNumber.from((1e18).toString()).mul(
            5000,
        );
        const usd0RepayAmt = usdoBorrowVal.add(usd0Extra); // Repays the full amount from another
↪   address. Borrows to eao1, repays from deployer
        await usd0.mint(deployer.address, usd0RepayAmt);
        await usd0.approve(yieldBox.address, usd0RepayAmt);
        await yieldBox
            .depositAsset(
                await wethBigBangMarket.assetId(),
                deployer.address,
                deployer.address,
                usd0RepayAmt,
                0,
            );
        await yieldBox
            .setApprovalForAsset(
                wethBigBangMarket.address,
                await wethBigBangMarket.assetId(),
                true,
            );
        await wethBigBangMarket.repay(
            deployer.address,
            deployer.address,
            false,
            userBorrowPart,
        );

        userBorrowPart = await wethBigBangMarket.userBorrowPart(
            eoa1.address,
        );
        expect(userBorrowPart.eq(0)).to.be.true;

        let collateralShares = await wethBigBangMarket.userCollateralShare(
                deployer.address,
                );
        expect(collateralShares.gt(0)).to.be.true;

        await wethBigBangMarket.removeCollateral(
                deployer.address,
                deployer.address,
                collateralShares,
            );

        collateralShares = await wethBigBangMarket.userCollateralShare(
                eoa1.address,
            );
        expect(collateralShares.eq(0)).to.be.true;

        borrowfeesDeployer = await wethBigBangMarket.openingFees(deployer.address);
        borrowfeesA = await wethBigBangMarket.openingFees(eoa1.address);
        console.log("UserA borrow fee after:", borrowfeesA);
        console.log("UserDeployer borrow fees after:",borrowfeesDeployer );

        let balanceAfter = await usd0.balanceOf(wethBigBangMarket.address);
        console.log("BBmarket usd0 balance:", balanceAfter);
    });
});
```

Likelihood: High + Impact: Medium = Severity: High.

Recommendation: Consider maintaining the same address while accounting for `openingFees` and `userBorrowPart`

Tapioca: Fixed in PR 292.

5.2.23 Non-standard ERC20 tokens will be stuck in TapiocaOFT

Severity: High Risk

Context: TOFTVault.sol#L40, Balancer.sol#L236

Description: Non-standard ERC20 tokens such as USDT implement a transfer function with no return value. This means that calling the transfer function on such a token will revert when it attempts to decode the boolean return value that is expected for standard ERC20s.

TapiocaOFT calls a transfer function expecting a boolean return value in `TOFTVault.withdraw()` and `Balancer.emergencySaveTokens()`, both of which will revert for tOFTs of non-standard ERC20s. Given that Singularity markets are meant to enable Tapioca to feature riskier collateral assets for attractive yields, it is conceivable that Tapioca would want to support the widely-used USDT as collateral in future.

Impact: Non-standard ERC20 collateral will be stuck in TapiocaOFT Vault (or Balancer) and cannot be withdrawn leading to loss of user deposits.

Likelihood: Medium + Impact: High = Severity: High.

Recommendation: Consider using OpenZeppelin's SafeERC20 which handles optional return values for ERC20 functions.

Tapioca: Fixed in PR 120.

5.2.24 Native TOFT wraps will always revert

Severity: High Risk

Context: BaseTOFT.sol#L561-L562, TOFTVault.sol#L32

Description: For native TOFTs, i.e. when `erc20 == address(0)`, `TapiocaOFT.wrap()` and `mTapiocaOFT.wrap()` end up calling `vault.depositNative()` but miss forwarding the received `msg.value`. This causes `vault.depositNative()` to revert at the check `if (msg.value == 0) revert ZeroAmount()`.

Impact: All native TOFT wraps will always revert.

Likelihood: High + Impact: Medium = Severity: High.

Recommendation: Change to:

```
  function _wrapNative(address _toAddress) internal virtual {
-     vault.depositNative();
+     vault.depositNative{value: msg.value}();
      _mint(_toAddress, msg.value);
  }
```

5.2.25 BaseTOFT `triggerSendFrom()` and `triggerSendFromWithParams()` will always revert

Severity: High Risk

Context: BaseTOFT.sol#L416-L429, BaseTOFT.sol#L473-L485, BaseTOFTGenericModule.sol#L44-L87, BaseTOFTGenericModule.sol#L201-L248

Description: `triggerSendFrom()` and `triggerSendFromWithParams()` pass an incorrect `Module.Options` argument to `_executeModule()`. These functions are actually implemented in `BaseTOFTGenericModule` and not in `BaseTOFTOptionsModule`. While the encoded data correctly uses `BaseTOFTGenericModule.triggerSendFromWithParams.selector` and `BaseTOFTGenericModule.triggerSendFrom.selector`, it should be executed on `Module.Generic` which implements these functions.

Impact: The `module.delegatecall(_data)` in `_executeModule()` will fail and cause a revert in the callers as well because `_forwardRevert == false` for these calls. As a result all tOFT `triggerSendFrom()` and `triggerSendFromWithParams()` will always revert.

Likelihood: High + Impact: Medium = Severity: High.

Recommendation: Replace `Module.Options` with `Module.Generic` in `BaseTOFT.triggerSendFrom()` and `BaseTOFT.triggerSendFromWithParams()`.

Tapioca: Fixed in PR 118.

5.2.26 Missing whitelist check in `leverageDownInternal()` allows stealing of native TOFT balance

Severity: High Risk

Context: BaseTOFTLeverageDestinationModule.sol#L162, USDOLeverageDestinationModule.sol#L134-L139, USDOLeverageDestinationModule.sol#L150-L151, BaseTOFTLeverageDestinationModule.sol#L171-L172, USDOLeverageDestinationModule.sol#L134-L135, USDOLeverageDestinationModule.sol#L110-L113, BaseTOFTLeverageDestinationModule.sol#L143-L147

Description: While leverage destination modules apply whitelist checks on `externalData.swapper`, they are missing similar whitelist checks on user-provided `externalData.tOft`, `externalData.magnetar`, `externalData.srcMarket` and `swapData.tokenOut` addresses.

Impact: While it is not clear how other user-provided addresses may be exploitable, the use of arbitrary `swapData.tokenOut` combined with user-controlled `airdropAmount` in `IUSDOBase(swapData.tokenOut).sendAndLendOrRepay{value: airdropAmount}()` call of `BaseTOFTLeverageDestinationModule.leverageDownInternal()` allows an attacker to steal all the balance of `mTapiocaOFT` tokens when their underlying tokens are native. This is similar to C4-1293.

Other addresses, including a similar issue that exists in `USDOLeverageModule.leverageUpInternal()`, allow attackers to make arbitrary external calls in the context of leverage destination modules, which at the very least may be used for griefing.

Likelihood: Medium + Impact: High = Severity: High.

Recommendation: Apply whitelist checks for all user-provided addresses on both source and destination chains.

Tapioca: Addressed in PRs 119 and 287.

5.2.27 Errors in `_yieldBoxShares` accounting will lead to incorrect and potential loss of user funds in crosschain yieldbox strategies

Severity: High Risk

Context: SGLLendingCommon.sol#L34, SGLLendingCommon.sol#L58-L63, SGLLendingCommon.sol#L92, SGLLiquidation.sol#L66-L70, SGLLiquidation.sol#L314-L323. SGLCommon.sol#L231, SGLCommon.sol#L262-L268, Singularity.sol#L225

Description: `_yieldBoxShares` is reportedly tracked for accounting in SGL to facilitate crosschain yieldbox strategies. However, there are errors/inconsistencies in its accounting where YB shares are not appropriately added/removed across the different SGL functions.

Impact: Errors in `_yieldBoxShares` accounting will lead to incorrect and potential loss of user funds in crosschain yieldbox strategies.

Likelihood: High + Impact: Medium = Severity: High.

Recommendation: Revisit `_yieldBoxShares` accounting to fix these errors across different functions appropriately.

Tapioca: I think we can remove `_yieldBoxShares` completely. It was added for cross chain yieldbox strategies that we don't have anymore.

Spearbit: Acknowledged that this entire logic is planned for removal.

Tapioca: Fixed; already removed in PR 274.

5.2.28 Incorrect application of elastic units across Liquidations

Severity: High Risk

Context: BBLiquidation.sol#L45, SGLLiquidation.sol#L47, BBLiquidation.sol#L208, SGLLiquidation.sol#L238

Description: Across the liquidation flow there are multiple instances where the userBorrowPart not converted to their elastic units, which could result in incorrect calculations causing inability to liquidate bad debts in some cases.

- Case 1: BBLiquidation.sol#L45, SGLLiquidation.sol#L47.

  In both the above instances the `borrowAmountWithBonus` is calculated directly using the `userBorrowPart` which is the base user part without converting it to its elastic value. Since `requiredCollateral` is still calculated using `borrowAmountWithBonus` which is not the elastic amount, there is a possibility this can be undervalued resulting in `liquidateBadDebt` to revert.

- Case 2: BBLiquidation.sol#L208, SGLLiquidation.sol#L238.

  The `collateralShare` here is calculated using the `borrowPartWithBonus` which is initially calcuated through `computeClosingFactor` at BBLiquidation.sol#L175 where the actual elastic amount is considered. However in the following lines based on the check at BBLiquidation.sol#189-L194 `borrowPartWithBonus` could be reallocated with a with the base values and not elastic. This could result in incorrect value of `collateralShare`.

Recommendation: Convert the above `base` part values into their respective `elastic` values in all the cases.

5.2.29 Fees will be lost due to incorrect `transfer`

Severity: High Risk

Context: USDO.sol#L101-L109

Description: `extractFees` is using the `public` `transfer` function which will cause the caller to pay for the tokens transferred (see USDO.sol#L101-L109):

```
function extractFees() external onlyOwner {
    if (_fees > 0) {
        uint256 balance = balanceOf(address(this));

        uint256 toExtract = balance >= _fees ? _fees : balance;
        _fees -= toExtract;
        if (!transfer(msg.sender, toExtract)) revert Failed(); /// @audit HIGH | All fees are lost
↪   forever
    } /// @audit NOTE: Caller is transfer to self
}
```

This will have the caller transfer to self, and will cause all fees to be lost.

Recommendation: Rewrite to (see USDO.sol#L107-L108):

```
if (_transfer(msg.sender, toExtract)) revert Failed();
```

Note that with the use of OZ, `_transfer` won't fail so it doesn't need to be checked.

Tapioca: Fixed it in PR 269.

Spearbit: Verified.

5.2.30 `BigBang.minDebtSize` sidestep causes `accrue` to revert, breaking a new market and the ETH market indefinitely

Severity: High Risk

Context: BBCommon.sol#L36-L37

Description: `BigBang.borrow` has a `debtStartPoint` which is the minimum debt per position (see BBBorrow.sol#L24-L36):

```
function borrow(
    address from,
    address to,
    uint256 amount
)
    external
    optionNotPaused(PauseType.Borrow)
    notSelf(to)
    solvent(from, false)
    returns (uint256 part, uint256 share)
{
    if (amount < debtStartPoint) revert NotEnough();
```

This is enforced only on `borrow` and not on `repay` nor on liquidations (since dynamic premium exists the debt size could be below the `debtStartPoint`). The `BigBang.ethMarket` tries to accrue all other markets and due to this, a new market can be bricked permanently by:

- Borrowing above `debtStartPoint`.
- Repaying to cause the total debt to be below `debtStartPoint`.

This will cause this line in the `getDebtRate` function to permanently revert (see BBCommon.sol#L36-L37):

```
uint256 debtPercentage = ((_currentDebt - debtStartPoint) *
```

Since `_accrue` relies on `getDebtRate` this will brick both the Secondary Market as well as the ETH Market.

Proof of concept: Add this test to your `bigBang.test.ts`:

```
it.only('Can bork the pools via the function', async () => {
    const {
        wethBigBangMarket,
        wbtcBigBangMarket,
        weth,
        wethAssetId,
        wbtc,
        wbtcAssetId,
        yieldBox,
        deployer,
        timeTravel,
        bar,
    } = await loadFixture(register);

    //borrow from the main eth market
    await weth.approve(yieldBox.address, ethers.constants.MaxUint256);
    await yieldBox.setApprovalForAll(wethBigBangMarket.address, true);

    const wethMintVal = ethers.BigNumber.from((1e18).toString()).mul(
        50,
    );
    await weth.updateMintLimit(wethMintVal.mul(100));
    await timeTravel(86401);
    await weth.freeMint(wethMintVal);
    const valShare = await yieldBox.toShare(
        wethAssetId,
        wethMintVal,
        false,
    );
    await yieldBox.depositAsset(
        wethAssetId,
        deployer.address,
        deployer.address,
        0,
        valShare,
    );
    await wethBigBangMarket.addCollateral(
        deployer.address,
        deployer.address,
        false,
        0,
        valShare,
    );

    const usdoBorrowVal = ethers.utils.parseEther('10000');
    await wethBigBangMarket.borrow(
        deployer.address,
        deployer.address,
        usdoBorrowVal,
    );

    let userBorrowPart = await wethBigBangMarket.userBorrowPart(
        deployer.address,
    );
```

```
    const ethMarketTotalDebt = await wethBigBangMarket.getTotalDebt();
    expect(ethMarketTotalDebt.eq(userBorrowPart)).to.be.true;

    const ethMarketDebtRate = await wethBigBangMarket.getDebtRate();
    expect(ethMarketDebtRate.eq(ethers.utils.parseEther('0.005'))).to.be
        .true;

    //wbtc market
    const initialWbtcDebtRate = await wbtcBigBangMarket.getDebtRate();
    const minDebtRate = await wbtcBigBangMarket.minDebtRate();
    expect(initialWbtcDebtRate.eq(minDebtRate)).to.be.true;

    await wbtc.approve(yieldBox.address, ethers.constants.MaxUint256);
    await yieldBox.setApprovalForAll(wbtcBigBangMarket.address, true);

    const wbtcMintVal = ethers.BigNumber.from((1e18).toString()).mul(
        50,
    );
    await wbtc.updateMintLimit(wbtcMintVal.mul(10));
    await timeTravel(86401);
    await wbtc.freeMint(wbtcMintVal.mul(5));
    const wbtcValShare = await yieldBox.toShare(
        wbtcAssetId,
        wbtcMintVal,
        false,
    );
    await yieldBox.depositAsset(
        wbtcAssetId,
        deployer.address,
        deployer.address,
        0,
        wbtcValShare,
    );
    await wbtcBigBangMarket.addCollateral(
        deployer.address,
        deployer.address,
        false,
        0,
        wbtcValShare,
    );

   const wbtcMarketusdoBorrowVal = ethers.utils.parseEther('2987');
   /// @audit Borrow above minDebtSize
    await wbtcBigBangMarket.borrow(
        deployer.address,
        deployer.address,
        wbtcMarketusdoBorrowVal,
    );

    userBorrowPart = await wbtcBigBangMarket.userBorrowPart(
        deployer.address,
    );

    const wbtcMarketTotalDebt = await wbtcBigBangMarket.getTotalDebt();
    expect(wbtcMarketTotalDebt.eq(userBorrowPart)).to.be.true;


    /// @audit Repay to drag totalDebt below minDebtSize
    await wbtcBigBangMarket.repay(
        deployer.address,
        deployer.address,
        true,
```

```
        wbtcMarketusdoBorrowVal.mul(99).div(100),
    );
    console.log("We can repay, less than 100% so we go below min")

    // Accrue should revert now due to this
    try {
        await wbtcBigBangMarket.accrue()
    } catch (e) {
        console.log("e", e)
        console.log("And we got the revert we expected")

    }

    try {
        // We cannot repay rest
        await wbtcBigBangMarket.repay(
            deployer.address,
            deployer.address,
            true,
            wbtcMarketusdoBorrowVal.mul(1).div(100),
        );
    } catch (e) {
        console.log("e", e)
        console.log("We cannot repay")
    }

    try {
        // We cannot borrow anymore due to accrue
        await wbtcBigBangMarket.borrow(
            deployer.address,
            deployer.address,
            wbtcMarketusdoBorrowVal,
        );
    } catch (e) {
        console.log("e", e)
        console.log("And we cannot borrow")
    }
})
```

Recommendation: The invariant of `minDebtSize` needs to be rethought, hence we would recommend assuming that `minDebt` will be 0. We also recommend making all hot paths resilient against underflows by adding invariant testing and proving that those functions will not revert due to an edge case.

Tapioca: Removed `debtStartPoint` in PR 262.

Spearbit: The fix will prevent reverts due to overflow due to this mechanism. It may still be possible for other overflows to cause reverts of `accrue`. We recommend extensive invariant testing is done to the `accrue` logic as a means to ensure that reverts will not happen on the live system.

5.2.31 Since advancing the epoch is permissionless the tapOFT DAO recovery can be griefed by anyone

Severity: High Risk

Context: AirdropBroker.sol#L419-L422

Description: TAP recovery can be subject to griefing as `newEpoch()` can be run by anyone past epoch `9`:

- AirdropBroker.sol#L419-L422

```
function daoRecoverTAP() external onlyOwner {
// see the line below
    require(epoch == 9, "adb: too soon");
    tapOFT.transfer(msg.sender, tapOFT.balanceOf(address(this)));
}
```

- AirdropBroker.sol#L308-L328

```
// see the line below
function newEpoch() external {
    if (block.timestamp < lastEpochUpdate + EPOCH_DURATION)
        revert TooSoon();

    // Update epoch info
    lastEpochUpdate = uint64(block.timestamp);
    // see the line below
    epoch++;
    // ...
    emit NewEpoch(epoch, epochTAPValuation);
}
```

Impact: tapOFT funds will be permanently frozen as there is no mechanics to reduce the epoch counter.

Per high asset freeze impact and medium probability (the griefing is very cheap, but provides no benefits for the attacker) setting the overall severity to be high.

Recommendation: Consider using `require(epoch >= 9, ...)` to unlink the rescue from the exact epoch number:

```
  function daoRecoverTAP() external onlyOwner {
-     require(epoch == 9, "adb: too soon");
+     require(epoch >= 9, "adb: too soon");
      tapOFT.transfer(msg.sender, tapOFT.balanceOf(address(this)));
  }
```

Tapioca: Fixed it in PR 115.

Spearbit: Fix looks ok.

5.3 Medium Risk

5.3.1 New lockers can grief starting epoch beneficiaries by frontrunning `newEpoch()`

Severity: Medium Risk

Context: TapiocaOptionBroker.sol#L280-L288, TapiocaOptionBroker.sol#L350-L360

Description: New locker can run `participate()` just before `newEpoch()` call, for example in the first block of a new epoch. Compared to running it just after `newEpoch()` it will make the only difference in `netDepositedForEpoch[old_epoch + 1] = netDepositedForEpoch[new_epoch]` being increased by `lock.ybShares`. It doesn't look to benefit the attacker by itself since the option exercise will require one full epoch anyway. But this will always dilute the reward base computation for the current lockers, griefing them. I.e. new locker can choose to grief current lockers by running `lock()` → `participate()` → `newEpoch()` first in the very beginning of new epoch time period. Their payoffs will not be lost, the additional outcome is `netDepositedForEpoch[new_epoch]` increase at some additional MEV cost for being the first to run `newEpoch()`.

`netDepositedForEpoch` increase in `participate()` is based on the current `epoch` reading (see TapiocaOptionBroker.sol#L350-L360):

```
// Record amount for next epoch exercise
// see the line below
netDepositedForEpoch[epoch + 1][lock.sglAssetID] += int256(
    uint256(lock.ybShares)
);
// see the line below
uint256 lastEpoch = _timestampToWeek(lock.lockTime + lock.lockDuration);
// And remove it from last epoch
// Math is safe, check `_emitToGauges()`
netDepositedForEpoch[lastEpoch + 1][lock.sglAssetID] -= int256(
    uint256(lock.ybShares)
);
```


It is then used as epoch reward base on exercise (see TapiocaOptionBroker.sol#L471-L479):

```
uint256 netAmount = uint256(
    // see the line below
    netDepositedForEpoch[cachedEpoch][tOLPLockPosition.sglAssetID]
);
if (netAmount <= 0) revert NoLiquidity();
uint256 eligibleTapAmount = muldiv(
    tOLPLockPosition.ybShares,
    gaugeTotalForEpoch,
    netAmount
);
```

Impact: current lockers can have their rewards diluted by `lock.ybShares` of a new locker, who can't exercise due to one epoch delay.

Likelihood: Medium (griefing cost is MEV related only; as argued in other issues whenever there is any additional benefit to running `newEpoch()` the first generally speaking it exceeds the related costs as usual, not related to any surface, benefit is small enough, so outbidding regular `newEpoch()` callers is not costly) + Impact: Medium (dilution is proportional to and so limited by the size of attacker's position) = Severity: Medium.

Recommendation: Since the period in the very beginning of each new epoch, before `newEpoch()` is run, does have `epoch` and timestamp mismatch, and the lockers who had position before could `participate()` before old epoch time period run out, while fresh lockers who just created a position will have only longer-term locker griefing as an impact, i.e. there is no valid use case for this situation, consider forbidding `participate()` whenever `_timestampToWeek(block.timestamp) > epoch`, e.g (TapiocaOptionBroker.sol#L280-L288):

```
  function participate(
      uint256 _tOLPTokenID
  ) external whenNotPaused returns (uint256 oTAPTokenID) {
      // Compute option parameters
      LockPosition memory lock = tOLP.getLock(_tOLPTokenID);
+     if (_timestampToWeek(block.timestamp) > epoch)
+         revert AdvanceEpochFirst();
      bool isPositionActive = _isPositionActive(lock);
      if (!isPositionActive) revert OptionExpired();

      if (lock.lockDuration < EPOCH_DURATION) revert DurationTooShort();
```

Tapioca: Addressed in PR 163.

Spearbit: Fix looks ok.

5.3.2 Late stakers in each epoch period can lose their first option payoff

Severity: Medium Risk

Context: TapiocaOptionBroker.sol#L450-L451, TapiocaOptionBroker.sol#L464-L465

Description: For the locker who entered late in the epoch, just before its end, the window to exercise the first option is limited to the similar end period during the next epoch, otherwise the exercise will not be possible. For example, if a user has entered during the last block of epoch, they can exercise first epoch option in the very last block of the next epoch only. If a user doesn't understand this limitation, the chances are high that they will lose first epoch payoff.

`exerciseOption()` prohibits exercise for one `EPOCH_DURATION` since locking (see TapiocaOptionBroker.sol#L464-L465):

```
if (block.timestamp < tOLPLockPosition.lockTime + EPOCH_DURATION)
    revert OneEpochCooldown(); // Can only exercise after 1 epoch duration
```


Also in order to exercise the position has to be active (see TapiocaOptionBroker.sol#L450-L451):

```
bool isPositionActive = _isPositionActive(tOLPLockPosition);
if (!isPositionActive) revert OptionExpired();
```

At the same time, `_isPositionActive()` controls for the epoch number. So for example, if a user locked for one epoch only in the last block of the previous epoch, their `_timestampToWeek(_lock.lockTime + _lock.lockDuration) = current_epoch` and they will be able to exercise during the current epoch only, i.e. once `newEpoch()` be run `_isPositionActive()` will be false and execution be denied (see TapiocaOptionBroker.sol#L620-L631):

```
function _isPositionActive(
    LockPosition memory _lock
) internal view returns (bool isPositionActive) {
    if (_lock.lockTime <= 0) revert PositionNotValid();
    if (_isSGLInRescueMode(_lock)) revert SingularityInRescueMode();
    // see the line below
    uint256 expiryWeek = _timestampToWeek(
        _lock.lockTime + _lock.lockDuration
    );

    isPositionActive = epoch <= expiryWeek;
}
```

Also see TapiocaOptionBroker.sol#L599-L603:

```
function _timestampToWeek(
    uint256 timestamp
) internal view returns (uint256) {
    return ((timestamp - emissionsStartTime) / EPOCH_DURATION);
}
```

This way in such a case they, while being locked for the full `EPOCH_DURATION`, will effectively have only one block for option execution. This and any other small enough window are barely reachable for a typical user.

More generally, users locking late in an epoch will have only the same remaining period of time (i.e. period from locking to the end of nearest epoch) in the next epoch for their first option execution, and otherwise it will be lost.

Impact: Users unaware of such limitation, which has a technical nature, can lose the payoff of the first option. It will not be redistributed to others as such users will have `netDepositedForEpoch` increased for the epoch and payoff funds be reserved for them, just being close to impossible to be claimed.

This would be the violation of the protocol logic that stipulates that number of options received by locker has to depend and be proportional to the time of locking. There are no limitations stated for the time of locking within epoch.

Likelihood: Low (assuming close to uniform distribution some share of the users will be locking close to the epochs ends) + Impact: High (one payoff is being frozen within the system) = Severity: Medium.

Recommendation: Consider introducing the minimum period that has to remain until epoch end for locking to be available. E.g. each last day of each one week epoch the locking will not be available with the explanation that otherwise it will be less than one day to claim first option payoff and the risk of losing it for good is too substantial. Also consider notifying in UI that only such a period is actually available for first option claiming (e.g. a user that locked 2 days before epoch end will have only 2 days in the end of the next epoch to execute, otherwise the right will be lost).

Tapioca: Instead of putting a fence smart contract wise, we'll display a warning on the frontend explaining that.

Spearbit: Acknowledged.

5.3.3 Liquidations will fail if oracles revert

Severity: Medium Risk

Context: BBLiquidation.sol#L100, C4-1026

Description: As described in C4-1026, liquidations will fail if oracles revert in their underlying calls. This issue does not appear to have been mitigated.

Impact: Liquidations may fail.

Likelihood: Low + Impact: High = Severity: Medium.

Recommendation: Consider using try-catch for oracle calls as recommended in C4-1026.

Tapioca: Addressed in PR 324.

5.3.4 Privileged roles and actions across Tapioca Market logic lead to centralization risks for users

Severity: Medium Risk

Context: Tapioca-bar

Description: There are several `onlyOwner` functions across Tapioca Market logic which affect critical protocol state and semantics. Some examples are highlighted below:

1. `Penrose.setCluster()` can change the whitelisting contract.
2. `Penrose.setBigBangEthMarketDebtRate()` can arbitrarily set `bigBangEthDebtRate`.
3. `Penrose.setConservator()` can change the `conservator` authorized to pause/unpause markets.
4. `Penrose.setUsdoToken()` can change the underlying USDO token.
5. `Penrose.registerSingularityMasterContract()` can register new Singularity master contracts.
6. `Penrose.registerSingularity()` can deploy and register new Singularity contracts.
7. `Market.setMarketConfig()` can set/change all market parameters and addresses.
8. `BaseUSDO.setMinterStatus()` can assign USDO minter privileges to anyone.
9. `BaseUSDO.setBurnerStatus()` can assign USDO burner privileges to anyone.
10. `BigBang.setAssetOracle()` can set/change asset oracle.

Impact: If any of the privileged roles are compromised, they can arbitrarily affect critical protocol-wide state and semantics.

Likelihood: Low + Impact: High = Severity: Medium.

Recommendation: Consider:

1. Documenting all the privileged roles and actions for protocol user awareness.
2. Enforcing role-based access control where different privileged roles control different protocol aspects and are backed by different keys to follow separation-of-privileges security design principle.
3. Enforcing reasonable thresholds and checks wherever possible.
4. Emitting events for all privileged actions.
5. Putting privileged actions affecting critical protocol semantics behind timelocks so that users can decide to exit/engage.
6. Following the strictest opsec guidelines for privileged keys e.g. use of reasonable multisig and hardware wallets.

Tapioca: Acknowledged.

Spearbit: Acknowledged.

5.3.5 `Vesting` Overflow Check can be bypassed

Severity: Medium Risk

Context: Vesting.sol#L178-L196

Description: `registerUsers` performs a check against overflow, by ensuring that the new total is not lower than the previous total

```
if (cachedTotalAmount > totalAmount) revert Overflow();
```

This is not sufficient, as it's possible for values to cause multiple overflows, which would allow setting `user.amounts` to values whose sum is greater than `type(uint256).max` while passing the constraint:

```
unchecked {
    uint256 len = _users.length;
    for (uint256 i; i < len; ) {
        // Checks
        if (_users[i] == address(0)) revert AddressNotValid();
        if (_amounts[i] == 0) revert AmountNotValid();
        if (users[_users[i]].amount > 0) revert AlreadyRegistered(); /// @audit Uniqueness is implicity
↪   in here

        // Effects
        data.amount = _amounts[i];
        users[_users[i]] = data;

        totalAmount += _amounts[i]; /// @audit Overflow of total amounts

        ++i;
    }

    // Record new totals
    if (cachedTotalAmount > totalAmount) revert Overflow();
```

The simplest example would be:

- Set values for which the sum(amounts) is exactly equal to `type(uint256).max`.
- Those values will pass the check, while giving to each user.amount some amount that is non-zero.

Normally the finding would be of higher severity, however, it relies on an admin mistake, warranting a lower severity.

Recommendation: Remove `unchecked` for the line `totalAmount += _amounts[i]`. Generally speaking, "local invariants" can be unchecked, while global invariants should be checked. Solmate ERC20 does a great job at showing this philosophy.

Tapioca: Fixed in PR 153.

5.3.6 `updateExchangeRate` integration with `seer` will result in incorrect `updated` value

Severity: Medium Risk

Context: Seer.sol#L75-L83

Description: From the `Seer` code we see that it will return `true` for `success` and not for an update

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

    (, uint256 high) = _readAll(inBase);
    return (true, high);
}
```

This will cause `updateExchangeRate` to incorrectly interpret the return value as an "update" while in reality that's a constant flag.

Recommendation: Consider instead returning the latest timestamp from Seer and verify it in `updateExchangeRate` as a means to more accurately implement staleness check.

Tapioca: Acknowledged. Seer is meant to use both Uni & CL as a source feed, and since Uni is always returning up to date value I don't think it's necessary to change that.

Spearbit: Consider specifying that:

- Based on them implementation of the TWAP the TWAP may be using a read that was generated at time X, meaning that it would be incorrect to categorize it as fresh at time `x + y`.
- Chainlink would also have an `updatedAt` time and that would arguably now be `now` at all time.

Tapioca: For context, we still do a bunch of checks on the timestamp so we kinda outsourced the safety measures from SGL/BB to the oracle, for example we're assuming that it should always return an up to date value, according to the staleness requirement-

Spearbit: Acknowledged.

5.3.7 TapiocaZ admin privilege: end user risks collection

Severity: Medium Risk

Context: TapiocaZ

- `Balancer.emergencySaveTokens` **carries admin risk to end users**
  - **Description:** Through this function (see Balancer.sol#L228-L238):

```
function emergencySaveTokens(
    address _token,
    uint256 _amount
) external onlyOwner {
    if (_token == address(0)) {
        (bool sent, ) = msg.sender.call{value: _amount}("");
        if (!sent) revert Failed();
    } else {
        if (!IERC20(_token).transfer(msg.sender, _amount)) revert Failed();
    }
}
```

    The `owner` could sweep any token in `Balancer` causing losses to end users.

- **Toggle balancers to deny service to user -** ***super edge case***
  - **Description:** mTapiocaOFT.sol#L107-L112

```
function wrap(
    address _fromAddress,
    address _toAddress,
    uint256 _amount
) external payable onlyHostChain {
    if (balancers[msg.sender]) revert NotAuthorized();
```

    mTapiocaOFT.sol#L123-L127

```
function unwrap(address _toAddress, uint256 _amount) external {
    if (!connectedChains[block.chainid]) revert NotHost();
    if (balancers[msg.sender]) revert NotAuthorized();
    _unwrap(_toAddress, _amount);
}
```

    The way this would be done would be as follows:

    * Set user as balancers.
    * Have their tx revert.
    * Remove users from balancers.

- **Removing a chain will make the crossChain OFT worth zero**
  - **Description:** The admin has the ability of removing connected chains. Doing so will cause any other chain beside the native one to cause a loss to people that bridged.

- **Recommendation:** Disclose the Admin Risks in your documentation

Tapioca: Acknowledged. We updated the code to only enable a chain, but not disable it:

```
/// @notice updates a connected chain whitelist status
/// @param _chain the block.chainid of that specific chain
function setConnectedChain(uint256 _chain) external onlyOwner {
    emit ConnectedChainStatusUpdated(_chain, connectedChains[_chain], true);
    connectedChains[_chain] = true;
}
```

The other two won't happen, but interesting edge case.

Spearbit: Acknowledged.

5.3.8 `BaseTOFTStrategyModule.retrieveFromStrategy` allows arbitrary `assetId` which allows operator to lose tokens on behalf of owner

Severity: Medium Risk

Context: BaseTOFTStrategyModule.sol#L106-L119

Description: Due to a non-strict check on `assetId`, through a donation, an approved attacker can trigger withdrawals of any asset from Yieldbox on behalf of the owner, as long as the owner has approve the attacker on `srcChain` and has made the `OFT` an Operator of Yieldbox on their behalf. `retrieveFromStrategy` checks allowance as follows (see BaseTOFTStrategyModule.sol#L106-L119):

```
function retrieveFromStrategy(
    address _from,
    uint256 amount,
    uint256 assetId,
    uint16 lzDstChainId,
    address zroPaymentAddress,
    bytes memory airdropAdapterParam
) external payable {
    //allowance is also checked on market
    if (_from != msg.sender) {
        if (allowance(_from, msg.sender) < amount)
            revert AllowanceNotValid();
        _spendAllowance(_from, msg.sender, amount);
    }
```

The parameter `assetId` is not validated, and assumed to be correct. Yieldbox permissions work as follows:

```
modifier allowed(address _from, uint256 _id) {
    _requireTransferAllowed(_from, isApprovedForAsset[_from][msg.sender][_id]);
    _;
}
// ...
function _requireTransferAllowed(address _from, bool _approved) internal view virtual {
    require(_from == msg.sender || _approved || isApprovedForAll[_from][msg.sender] == true, "Transfer
↪   not allowed");
}
```

In the case in which the tOFT is made operator of the yieldbox for the cross chain user, any other user given allowance will have the ability of burning other tokens amounts on their behalf.

Proof of concept: This example uses USDO as the "real" token:

- Victim approves tOFT as Operator on `dstChain`.
- Victim grants allowance to Attacker on `srcChain`.
- Attacker donates `amount` of USDO to tOFT in `dstChain`.

- Attacker calls `retrieveFromStrategy` with an `assetId` that corresponds to a high value position.
- tOFT will withdraw from Yieldbox and keep those tokens stuck in its contract.
- tOFT will use the USDO from the donation to continue.

Recommendation: Use `chainId` → `assetId` from mappings as a means to ensure only the correct `assetId` can be withdraw xChain.

Tapioca: `retrieveFromStrategy` doesn't exist anymore.

5.3.9 New ETH market rate `Penrose` sets can be applied backwards

Severity: Medium Risk

Context: Penrose.sol#L273-L279

Description: ETH market interest rate can be changed via `onlyOwner setBigBangEthMarketDebtRate()`, whose logic does not apply the old rate beforehand (see Penrose.sol#L273-L279):

```
/// @notice sets the main BigBang market debt rate
/// @dev can only be called by the owner
/// @param _rate the new rate
function setBigBangEthMarketDebtRate(uint256 _rate) external onlyOwner {
    // see the line below
    bigBangEthDebtRate = _rate;
    emit BigBangEthMarketDebtRate(_rate);
}
```

But the rate is used in `_accrue()` for the period since last known update, `lastAccrued` (see BBCommon.sol#L74-L94):

```
function _accrue() internal override {
    IBigBang.AccrueInfo memory _accrueInfo = accrueInfo;
    // Number of seconds since accrue was called
    // see the line below
    uint256 elapsedTime = block.timestamp - _accrueInfo.lastAccrued;
    if (elapsedTime == 0) {
        return;
    }
    //update debt rate
    // see the line below
    uint256 annumDebtRate = getDebtRate();
    _accrueInfo.debtRate = uint64(annumDebtRate / 31536000); //per second
    _accrueInfo.lastAccrued = uint64(block.timestamp);

    Rebase memory _totalBorrow = totalBorrow;

    // Calculate fees
    uint256 extraAmount = 0;
    extraAmount =
        (uint256(_totalBorrow.elastic) *
            // see the line below
            _accrueInfo.debtRate *
            elapsedTime) /
        1e18;
```

Also see BBCommon.sol#L24-L25:

```
function getDebtRate() public view returns (uint256) {
    if (isMainMarket) return penrose.bigBangEthDebtRate(); // default 0.5%
```

Impact: ETH market will use new interest rate for the old period, from the last `_accrue()` call to the new interest rate setting block, i.e. the interest accrual for the period will be incorrect.

Likelihood: Low + Impact: High = Severity: Medium.

Recommendation: Consider to accrue main ETH market first, so previous period will be covered with the prevailing old value of the rate, e.g. in Penrose.sol#L273-L279:

```
  /// @notice sets the main BigBang market debt rate
  /// @dev can only be called by the owner
  /// @param _rate the new rate
  function setBigBangEthMarketDebtRate(uint256 _rate) external onlyOwner {
+     IBigBang(bigBangEthMarket).accrue();
      bigBangEthDebtRate = _rate;
      emit BigBangEthMarketDebtRate(_rate);
  }
```

Tapioca: Already fixed in PR 350.

Spearbit: Fix looks ok.

5.3.10 Non `CEI` Conformity allows reentrancy before applying `revokes`

Severity: Medium Risk

Context: BaseTOFTGenericModule.sol#L44-L45

Description: `executSendFromWithParams` allows for `unwrap`ing of Ether, this will use a `call` with uncapped gas, which may be used by the recipient to reenter before `revokes` are applied.

This could similarly happen for ERC20s that have hooks on transfer and notify the recipient of a transfer (BaseTOFTGenericModule.sol#L90-L147):

```
function executSendFromWithParams( /// @audit QA: Typo
    address,
    uint16 lzSrcChainId,
    bytes memory,
    uint64,
    bytes memory _payload
) public {
    // ...
    if (unwrap) {
        ITapiocaOFTBase tOFT = ITapiocaOFTBase(address(this));
        address toftERC20 = tOFT.erc20();

        tOFT.unwrap(address(this), amount);

        if (toftERC20 != address(0)) {
            IERC20(toftERC20).safeTransfer(toAddress, amount);
        } else {
            (bool sent, ) = toAddress.call{value: amount}(""); /// @audit Reentrancy here
            if (!sent) revert Failed();
        }
    }

    if (revokes.length > 0) {
        _callApproval(revokes, PT_SEND_FROM_PARAMS);
    }

    emit ReceiveFromChain(lzSrcChainId, toAddress, amount);
}
```

Due to this, the `revokes` may be ineffective as the caller may be able to spend more of them before the call returns to the `tOFT` context:

- Call `executSendFromWithParams`.
- Approves are set.
- Operations → unwrap.
- Refund is called, malicious receiver spends more of the allowance as they see fit.
- Revokes are performed.

Because of this, it would be best to swap the order of operations, to first revoke and then transfer the tokens. This has no particular additional risk but removes the possibility of a malicious receiver re-gaining execution control mid transaction.

Recommendation: Change the order of operations to:

- Approves are set.
- Operations → unwrap.
- Revoke.
- Transfer funds.

Tapioca: Fixed in PR 154.

5.3.11 `BaseTOFTMarketDestinationModule.remove` uses incorrect rounding direction for computing `shares` removing less than `amount` due to rounding

Severity: Medium Risk

Context: BaseTOFTMarketDestinationModule.sol#L219-L223

Description: `BaseTOFTMarketDestinationModule.remove` aims to remove `amount` from Yieldbox, but it uses this formula (see BaseTOFTMarketDestinationModule.sol#L219-L223):

```
uint256 share = IYieldBoxBase(ybAddress).toShare(
    assetId,
    removeParams.amount,
    false
);
```

This formula is meant to compute the `shares` issues on `deposit`. When calculating the `shares` needed for a `withdrawal`, the code should `roundUp`. This will cause, in scenarios in which rounding makes a difference, that withdrawal of an amount of `shares` that corresponds to a lower `amount` of tokens.

This may cause issues to integrator or cause xChain operations to be delayed, but should not cause a loss of value.

Recommendation: Change the code to

```
uint256 share = IYieldBoxBase(ybAddress).toShare(
    assetId,
    removeParams.amount,
    true
);
```

Tapioca: Fixed by rounding up `shares`, as done in PR 152.

Spearbit: Verified.

5.3.12 Incorrect `TapOFT._getChainId()` implementation may prevent minting and DSO emission of TAP tokens

Severity: Medium Risk

Context: TapOFT.sol#L280-L285, TapOFT.sol#L141-L149, TapOFT.sol#L217-L218, BaseUSDOStorage.sol#L81-L83, twTAP.sol#L648-L652

Description: Unlike `BaseUSDOStorage._getChainId()` which returns LayerZero's chain identifier `ILayerZeroEndpoint(lzEndpoint).getChainId()`, `TapOFT._getChainId()` incorrectly returns the EVM chain identifier `block.chainid`. These are completely different as noted in LayerZero Documentation: "*chainId values are not related to EVM ids. Since LayerZero will span EVM & non-EVM chains the chainId are proprietary to our Endpoints.*" For example, Arbitrum's LayerZero identifier is 110 while its EVM `chainID` is 42161.

As commented, `governanceChainIdentifier` is supposed to be the "*/// @notice LayerZero governance chain identifier*" which is set in the constructor and also via a setter `setGovernanceChainIdentifier()`. While minting and DSO emitting of TAP tokens, this identifier is compared against `_getChainId()` which should always fail if `governanceChainIdentifier` is set to LayerZero's chain identifier for say Arbitrum.

Impact: Incorrect `TapOFT._getChainId()` implementation will prevent minting and DSO emission of TAP tokens.

Likelihood: Low (this may be incorrectly set to EVM identifier only to pass this incorrect check) + Impact: High (failure of minting and DSO emitting of TAP tokens) = Severity: Medium.

This incorrect implementation is also present in `twTAP.sol` but that does not appear to be used anywhere.

Recommendation: Consider changing `TapOFT._getChainId()` implementation to return `ILayerZeroEndpoint(lzEndpoint).getChainId()`.

Tapioca: Addressed in PR 144.

5.3.13 A malicious `twTAP` owner can steal the TapOFT ETH balance while exiting cross-chain position

Severity: Medium Risk

Context: BaseTapOFT.sol#L339-L377, BaseTapOFT.sol#L411-L417, BaseTapOFT.sol#L294-L302, BaseTOFTOptionsDestinationModule.sol#L176-L188, BaseTapOFT.sol#L444-L447

Description: Cross-chain operations that trigger a callback from the destination chain to the source chain via `sendFrom` require the user to specify and send the gas required for the callback from the destination chain. This is typically done by specifying a "airdropAmount" in the encoded `lzPayload` payload which gets used as the gas amount during the callback in the destination function, for e.g. `ISendFrom(address(rewardTokens[i])).sendFrom{value: rewardClaimSendParams[i].ethValue}(...)` and `ISendFrom(tapSendData.tapOftAddress).sendFrom{value: airdropAmount}(...)`.

While this is done in most places, it is missed in `_unlockTwTapPosition()` which makes the call `this.sendFrom{value: address(this).balance}(...)` using the TapOFT ETH balance.

Impact: A malicious `twTAP` owner can steal the TapOFT ETH balance while exiting position where any excess TapOFT ETH balance beyond that required for gas is sent to the user-controlled refund address provided in `twTapSendBackAdapterParams.refundAddress`.

This is similar to C4-1290 but the value at risk is only the TapOFT ETH balance and not all the underlying balance of a native TOFT. TapOFT should only have ETH airdropped for LZ gas usage but this may be a non-trivial amount given that there is an explicit `rescueEth()` function to reclaim any unused ETH from the contract, which may be lost to a malicious `twTAP` owner.

Likelihood: Medium (Requires a malicious `twTAP` owner) + Impact: Medium (Loss of any TapOFT ETH balance) = Severity: Medium.

Recommendation: Like other similar cross-chain operations, consider sending/extracting the "airdropAmount" from the appropriate `adapterParams` to be then used as `this.sendFrom{value: airdropAmount}(...)` in the callback within the destination function. Do not use `this.sendFrom{value: address(this).balance}(...)`.

Tapioca: Addressed in PR 143.

5.3.14 BigBang ETH market liquidations change its total debt, but do not notify linked BB markets, making their interest rates potentially stale

Severity: Medium Risk

Context: BBLiquidation.sol#L43, BBLiquidation.sol#L108

Description: ETH market liquidation reduces the total debt of it, `IBigBang(penrose.bigBangEthMarket()).getTotalDebt()`, while interest rate logic of the other BB markets depends on it (see BBCommon.sol#L24-L45):

```
function getDebtRate() public view returns (uint256) {
    if (isMainMarket) return penrose.bigBangEthDebtRate(); // default 0.5%
    if (totalBorrow.elastic == 0) return minDebtRate;

    uint256 _ethMarketTotalDebt = IBigBang(penrose.bigBangEthMarket())
        .getTotalDebt();
    uint256 _currentDebt = totalBorrow.elastic;
    // see the line below
    uint256 _maxDebtPoint = (_ethMarketTotalDebt *
        debtRateAgainstEthMarket) / 1e18;

    if (_currentDebt >= _maxDebtPoint) return maxDebtRate;

    uint256 debtPercentage = ((_currentDebt - debtStartPoint) *
        DEBT_PRECISION) / (_maxDebtPoint - debtStartPoint);
    uint256 debt = ((maxDebtRate - minDebtRate) * debtPercentage) /
        DEBT_PRECISION +
        minDebtRate;

    if (debt > maxDebtRate) return maxDebtRate;

    return debt;
}
```


`reAccrueBigBangMarkets()` needs to be called in order to propagate the current ETH market total debt value to all the linked markets but it's not done at the moment.

Impact: linked markets can use the outdated interest rates, in which case they will accrue at the incorrect rate between the previous `_accrue()` calling operation there and ETH market liquidation (i.e. on the next linked market `_accrue()` the new ETH market total debt will be used backwards to the period prior to the liquidation).

Likelihood: Medium + Impact: Medium = Severity: Medium.

Notice, that `reAccrueBigBangMarket()` goes over all the linked markets, so it adds a substantial enough number of operations to the liquidation, so, in general, it's a trade-off between easing ETH market liquidation process by lowering its gas cost and having correct interest rate accruals across the linked BB markets.

Recommendation: Since the gas costs do not pose a substantial concern on Arbitrum, which is the target chain for all Big Bang markets as of now, and due to its special nature there will be only a very limited number of BB markets, consider calling linked markets synchronization on liquidations before changing the ETH market state (see BBLiquidation.sol#L29-L43):

```
  function liquidateBadDebt(
      // ...
  ) external onlyOwner {
      (bool updated, uint256 _exchangeRate) = oracle.get(oracleData);
      // ...
      if (_exchangeRate == 0) revert ExchangeRateNotValid();

      _accrue();
+     penrose.reAccrueBigBangMarkets();
```

And also BBLiquidation.sol#L87-L108:

```
  function liquidate(
      // ...
  ) external optionNotPaused(PauseType.Liquidation) {
      // ...

      // Oracle can fail but we still need to allow liquidations
      (bool updated, uint256 _exchangeRate) = oracle.get(oracleData);
      // ...
      if (_exchangeRate == 0) revert ExchangeRateNotValid();

      _accrue();
+     penrose.reAccrueBigBangMarkets();
```

Tapioca: Addressed in PR 319 and PR 328.

5.3.15 `Balancer.addRebalanceAmount()` allows owner to arbitrarily increase the rebalanceable amount which may cause rebalancing to fail

Severity: Medium Risk

Context: Balancer.sol#L281, Balancer.sol#L196-L209

Description: The initial expectation is that `Balancer.addRebalanceAmount()`, `Balancer.checker()` and `Balancer.rebalance()` will either be triggered manually or via Gelato automation. Ideally, `Balancer.addRebalanceAmount()` should be called to increase the rebalanceable amount whenever the `_srcOft` is wrapped to increase its underlying's balance. However, there is no sanity check to ensure that the updated `connectedOFTs[_srcOft][_dstChainId].rebalanceable` has sufficient underlying tokens because `Balancer.addRebalanceAmount()` allows owner to accidentally increase the rebalanceable amount by an arbitrary value.

Impact: `Balancer.rebalance()` called with `Balancer.checker()` payloads will revert if there isn't sufficient underlying tokens in the call to `ITapiocaOFT(_srcOft).extractUnderlying(_amount)`. This will continue to fail because there is no function to reduce `connectedOFTs[_srcOft][_dstChainId].rebalanceable` amount to make up for any accidental increases.

Likelihood: Low (`Balancer.addRebalanceAmount()` is called with an incorrect `_amount` value) + Impact: High (Rebalancing, which is a critical MetaTOFT functionality, will continue to fail) = Severity: Medium.

Recommendation: Add a sanity check in `Balancer.addRebalanceAmount()` to ensure for e.g. that `connectedOFTs[_srcOft][_dstChainId].rebalanceable <= ITapiocaOFT(_srcOft).vault.viewSupply()` after updation.

Tapioca: Addressed in PRs 149 and 165.

5.3.16 `collateralizationRate` can be set to a value that blocks all the liquidations

Severity: Medium Risk

Context: Market.sol#L256-L266, Market.sol#L305-L312

Description: There is no check for `collateralizationRate * (1 + liquidationMultiplier) < 1` and `computeClosingFactor()` can become permanently reverting, blocking the liquidations (see Market.sol#L305-L312):

```
//compute numerator
uint256 numerator = borrowPart - liquidationStartsAt;
//compute denominator
uint256 diff = (collateralizationRate *
    ((10 ** ratesPrecision) + liquidationMultiplier)) /
    (10 ** ratesPrecision);
// see the line below
int256 denominator = (int256(10 ** ratesPrecision) - int256(diff)) *
    int256(1e13);
```

This is C4-1012.

Likelihood: Low + Impact: High = Severity: Medium.

Recommendation: Consider adding the check, e.g. Market.sol#L256-L266:

```
  if (_collateralizationRate > 0) {
      require(
          _collateralizationRate <= FEE_PRECISION,
          "Market: not valid"
      );
      require(
          _collateralizationRate <= liquidationCollateralizationRate,
          "Market: collateralizationRate too big"
      );
+     require(
+         _collateralizationRate * (10 ** FEE_PRECISION + liquidationMultiplier) < 10 ** (2 *
↪   FEE_PRECISION),
+         "Market: CR * (1 + LM) >= 1"
+     );
      collateralizationRate = _collateralizationRate;
  }
```

Tapioca: Addressed in PR 317.

Spearbit: Fix looks ok.

5.3.17 `mtOFT` wrapping different Bridged Tokens introduced multiple Systemic Risks

Severity: Medium Risk

Context: Global scope

Description: mtOFT are meant to wrap "different tokens" that share some base token on some chain.

For example, they are meant to unify `wstETH` from Mainnet to be represented by the same `mtOFT` on OP, Arbitrum, etc..

However, `opwstETH` and `arbwstETH` are very different tokens, because they inherit the risks of the bridges from which they are minted (lock on Mainnet the "real" token, mint on `OP` the "receipt" token).

Due to this `mtOFT` expose themselves to systemic risks as they are inherently adopting the risks from all tokens that are wrappable via the same `mtOFT`

For example:

- Some tokens can have different decimals (USDT).
- Some tokens may have drastically different liquidities and spot values (OP vs ARB vs Mainnet for wstETH).
- Some tokens may no longer be bridgeable and may be depegged due to it (FTM).

If you allow unwrapping from ChainA to ChainB

Then:

- If the token in ChainB is overpriced, a clear arb is available and MEVers will always withdraw from ChainB to gain extra value.
- If the token in ChainB is underpriced, it will never be unwrapped.
  - This may be done with the goal of "forcing you" to rebalance.

Any rebalance operation is causing a loss to all users:

- If we assume a maximum slippage paid (rational since this is a security review).
- Then any rebalancing operation is causing a loss of slippage to all people.
- The loss will cause a "duration risk loss" to people, as their receipt tokens may not be redeemable.

Rebalancing may make it so that liquidations, that normally would be profitable are no longer possible:

- If the wrapping idea is valid, and some loss is possible.
- If the wrapped tokens are the assets from Pools.
- Then liquidations profitability will be based on available rebalanced liquidity.

Via the following considerations, we can formulate the following Medium Severity Findings:

- **Medium Severity:** All mtOFT deposits are subject to losses due to rebalancing.

  As we can see in the code, bridging costs up to 1%. We can assume that any hardcoded slippage may be brought up to that value. That would mean an instant loss of 1% of all assets sent to other chains. In the worst case scenarios that would be a 1% instant loss of the value of all deposits.

- **Medium Severity:** mtOFT rebalancing opens up to xChain arbitrage opportunity at the detriment of every other depositor.

  Any time one of the tokens that is `unwrappable` is more valuable than the token that is `wrappable`. An obvious arbitrage opens up. Which will cause all of the `unwrappable` tokens to be redeemed as they will offer a premium at basically no cost to the claimer.

- **Medium Severity:** mtOFT automated rebalancing gives a direct button to leak of value via xChain arbitrage.

  In lack of fees for wrapping and unwrapping, any automated rebalancing operation may be triggered by MEV exploiters, with the sole goal of having the protocol / other deposits pay bridging fees that would allow said MEV actors to claim tokens at a discount. Due to the high complexity of fairly pricing bridge risk, it may be best to never automatically rebalancing mOFTs as any automated and immutable logic could be exploited to leak value over time.

- **Medium Severity:** Liquidations may not be possible for mtOFT that have been `rebalanced` due to liquidity crunch.

  Due to xChain rebalancing, a liquidatable position may not be fully `unwrappable`. This would force a liquidator to be forced to hold the mtOFT and redeem it on another chain, which would expose them to:

  - Additional smart contract risk.
  - Currency risk (no risk free arb / need to edge their position).
  - Additional bridging duration risk / liquidity risk.
  - Additional bridging fees.

  Which can impact the profitability of liquidations, making it less likely than a liquidator will perform their market function.

Local fee Findings

These two findings are inspired by the discussion around adding `wrap`ing fees. We define "local pricing" as the cost of swapping from token to numeraire on chain X.

- **Medium Severity:** Static Wrapping Fee will not protect the system against Black Swan and High Variance Events.

  A static fee is a great start to ensure that in the vast majority of cases, local pricing doesn't introduce arbitrage. However, average values are not "all values", it's worth considering that when modeling an immutable or "slow to change" fee. Instead of looking at average values, it's important to look at edge case values as otherwise, the fee will "work until it doesn't".

- **Medium Severity:** Price Feeds may end up not behaving as intended.

  In order to evaluate past or current fees, Price Feed may be used. However, in the lack of a clear business agreement with a provider, such as Chainlink, price feeds should not be considered as a reliable way to capture "local pricing". I don't believe there's any indicator as to whether a Price Feed should be based on "local pricing" or whether it should report the "real price" of an asset. A salient example is what recently happened with Silo Finance.

  Silo was using wstETH/ETH as a means to ensure a fair "real" pricing of wstETH, and instead, they receive a price feed aggregate value that was a mixture of "local pricing" (volume-based) data as well as "real price". As of today, there doesn't seem to be a data provider that offers the exact values ("tamper-resistant local pricing of token pairs") that would be suitable to compute dynamic fees in any reliable way.

Contradictory findings

The following are 2 valid observations that arise from the chosen design, as discussed these problems are the edge of DeFi and seem to be unsolved as of now:

- **Medium Severity:** Lack of fees in wrapping and unwrapping is exposing tOFT to cross-chain arbitrage.

  As shown above, local prices may be different for "the same token", in lack of fees, the only cost for arbing these discrepancies is the L0 gas fee as well as some time/opportunity cost. This would make the system vulnerable to "skim arbitrages". A fee could prevent this

- **Medium Severity:** tOFT wrap / unwrap fees will reduce the profitability of liquidators, making it less likely that a liquidation will happen.

  Because fees are a cost to Liquidators, and most liquidators will prefer atomic arbitrages (start with Numeraire, wrap into Debt asset, liquidate, swap back into numeraire), wrapping and unwrapping fees will reduce the profitability of said liquidations. This may cause scenarios in which a liquidation is profitable but nobody is willing to perform it.

  This risk has to be kept into account if fees are to be introduced as a means to prevent cross-chain skim arbitrages.

- **Low Severity:** YieldBox Shares will have different values in different chains.

  Due to using Yieldbox as a foundational component for BB and SGL, any new chain will have a fresh Yieldbox Deployment, which may enable new opportunities for rebasing Yieldbox (as the local total supply of the mOFT token could be very low). This may also make it more difficult for end users to determine the value of certain positions as a share of an mOFT may mean different things on different chains.

Recommendation: Further explore the economic soundness of using mtOFTs as they seem to expose the protocol to many additional risks.

5.3.18 Rebalancing executed with `checker()` payloads will always fail for non-native TOFTs

Severity: Medium Risk

Context: Balancer.sol#L149-L154, Balancer.sol#L218, Balancer.sol#L392-L395

Description: The Balancer contract provides a helper `checker()` function to determine if a `rebalance()` can be performed for a particular `_srcOft` and `_dstChainId` combination, and also provides the `execPayload` required. However, it incorrectly sets the `ercData` to the `connectedOFTs` source + destination pool IDs for native TOFTs instead of non-native TOFTs by checking `ITapiocaOFT(_srcOft).erc20() == address(0)`. Stargate Pool IDs are required for non-native ERC20 tokens.

Impact: Rebalancing executed with `checker()` payloads will always fail for non-native TOFTs when `_sendToken()` attempts to ABI decode empty `_ercData` into source + destination pool IDs.

Likelihood: Low (Unclear if `rebalance()` is always called using `checker()` payloads) + Impact: High (rebalancing is critical to MetaTOFTs functionality) = Severity: Medium.

Recommendation: The conditional check in `checker()` should be:

```
- if (ITapiocaOFT(_srcOft).erc20() == address(0)) {
+ if (ITapiocaOFT(_srcOft).erc20() != address(0)) {
      ercData = abi.encode(
          connectedOFTs[_srcOft][_dstChainId].srcPoolId,
          connectedOFTs[_srcOft][_dstChainId].dstPoolId
      );
  }
```

Tapioca: Addressed in PR 146.

5.3.19 Disconnecting a chain will prevent existing wrapped MetaTOFTs from being unwrapped to their underlying and result in lock of user tokens

Severity: Medium Risk

Context: mTapiocaOFT.sol#L135-L145, mTapiocaOFT.sol#L120-L127

Description: MetaTOFTs may be unwrapped to their underlying tokens on any of the connected chains. However, the owner may arbitrarily disable the `connectedChains` whitelist status for any chain using `updateConnectedChain()`, which will prevent MetaTOFTs from being unwrapped on that chain.

Impact: Disconnecting a chain will prevent existing wrapped MetaTOFTs from being unwrapped to their underlying and resulting in lock of user tokens on that chain, thereafter requiring users to unwrap on other connected chains if any. If this is accidentally triggered by the owner then it will result in a temporary lock of user tokens on the disconnected chain, but if this is maliciously done across all connected chains of a particular MetaTOFT then it will result in lock/loss of user funds.

Likelihood: Low (Conditional on a compromised owner) + Impact: High (Lock/Loss of user funds in the worst case) = Severity: Medium.

Recommendation:

1. Specify the conditions under which an owner is allowed to disable the whitelist status of a connected chain.
2. Prevent owner from disconnecting the host chain to always allow unwrapping on it.
3. Add logic to check if there are existing wrapped assets on the chain being disconnected and evaluate options for users to rescue their underlying tokens before disabling the chain's whitelist status.
4. Consider adding a timelock for such critical functions to allow users to react by unwrapping their tokens.

Tapioca: Addressed in PR 143. You can only add connected chains now.

5.3.20 Incorrect setting of shared decimals will affect cross-chain token transfers and amount conversions

Severity: Medium Risk

Context: BaseTOFTStorage.sol#L82, LayerZero Documentation, OFTV2.sol#L12-L16

Description: LayerZero has a concept of "shared decimals" which is documented as:

> Shared Decimals is used to normalize the data type difference across EVM chain and non-Evm. Non-evm chains often has a Uint64 data type which limits the decimals of the token to a lower amount. Shared Decimals accounts for this and translates the higher decimals of EVM to lower decimals of non-evm. `shared decimals` should be set lower than 8 if you want a larger maximum send amount.

with the below guidance for EVM-only usage:

> If your tokens are only deployed on EVM chains and all have decimals larger than 8, it should be set as 8. For example, your tokens on all EVM chains have decimals of 18, the shared decimals on all chains should be set as 8.

However, TOFTs unconditionally initialize this value to `_decimal / 2` in the constructor for `OFTV2`, where `_decimal` is the number of decimals of the TOFT. This sets an incorrect value of `9` for typical ERC20 tokens with 18 decimals. For tokens such as USDC and USDT which have 6 decimals on Ethereum, this likely sets an unexpected value as well. Also, USDC and USDT have 18 decimals on other EVM chains such as BSC. It is not specified as to how these considerations should be managed for TOFTs to work as expected with LayerZero assumptions.

Impact: Incorrect setting of shared decimals will affect cross-chain token transfers and amount conversions, resulting in unexpected minimum/maximum thresholds for cross-chain token transfers and their conversion amounts.

Likelihood: High + Impact: Low (specific implications are undetermined given lack of further documentation and LayerZero internals being out-of-scope for this review) = Severity: Medium.

Recommendation: Evaluate decimals of all the tokens and chains under protocol consideration to determine the correct values of LayerZero shared decimals to be set appropriately for the different tokens.

Tapioca: Acknowledged. We are aware, I think this will change with V2 migration.

Spearbit: Acknowledged.

5.3.21 Cross-chain `exerciseOption()` will always fail when the user-provided `paymentToken` is not the TOFT or USDO

Severity: Medium Risk

Context: BaseTOFTOptionsModule.sol#L81-L100, BaseTOFTOptionsDestinationModule.sol#L91-L101, BaseTOFTOptionsDestinationModule.sol#L156-L174, TapiocaOptionBroker.sol#L442-L502, TapiocaOptionBroker.sol#L661-L667, TapiocaOptionBroker.sol#L553-L564, USDOOptionsModule.sol#L30-L111

Description: Tapioca allows users to exercise their TAP options using any of the accepted payment tokens. Cross-chain exercising of options aims to expose the same functionality via TOFTs.

Cross-chain `exerciseOption()` debits and credits `optionsData.paymentTokenAmount` of TOFTs on the source and destination chains respectively. While it allows users to specify `optionsData.paymentToken`, the implementation makes an incorrect assumption that this is always the same as the TOFT, which need not be the case. When the destination chain makes a call to `ITapiocaOptionsBroker(target).exerciseOption()`, that function attempts to transfer the payment tokens from the calling TOFT contract. If the user has specified a `optionsData.paymentToken` different from the TOFT, which is reasonable given the support for multiple payment tokens in the protocol, the transfer will fail because the TOFT contract does not have the required balance of payment tokens.

Impact: Cross-chain `exerciseOption()` will always fail when the user-provided `paymentToken` is not the TOFT.

Likelihood: Medium (`optionsData.paymentToken` needs to be different from the TOFT) + Impact: Medium (A primary cross-chain functionality fails) = Severity: Medium.

A similar scenario applies to `USDOOptionsModule.exerciseOption()` and USDO as well.

Recommendation: Add the required support to enable payment tokens other than TOFT/USDO as expected.

Tapioca: Created PRs 164, 141 and 315.

5.3.22 `Singularity` interest rate changes are drastically different based on compounded accruals

Severity: Medium Risk

Context: SGLCommon.sol#L124-L147

Description: The `accrue` math is meant to change the interest rate to a specific value over time

The formula chosen has the following properties:

- Equilibrium is found in having `utilization == minimumTargetUtilization`.
- When `minimumTargetUtilization > utilization` the interest rate decreases down to the minimum.
- When `maximumTargetUtilization < utilization` the interest rate continously increases.

This means that:

- An x%, with `x > maximumTargetUtilization` doesn't result in a constant rate.
- Over time, the rates can increase up to `maximumInterestPerSecond`.

Through testing, we have found that based on the frequency of accruals, rates can change dramatically faster than when no accruals happen.

Proof of Concept:


```
// SPDX-License Identifier: MIT

pragma solidity 0.8.17;

import "forge-std/Test.sol";
import "forge-std/console2.sol";


contract DemoInterestRate {

    uint256 minimumInterestPerSecond;
    uint256 maximumInterestPerSecond;

    uint256 minimumTargetUtilization;
    uint256 maximumTargetUtilization;

    uint256 internal constant FULL_UTILIZATION = 1e18;
    uint256 internal constant UTILIZATION_PRECISION = 1e18;
    uint256 internal constant FACTOR_PRECISION = 1e18;

    uint256 public currentInterestPerSecond;

    uint256 interestElasticity = 28800e36; // 8 hours ?


    function setMinInterest(uint256 newMin) external {
        minimumInterestPerSecond = newMin;
    }
    function setMaxInterest(uint256 newMax) external {
        maximumInterestPerSecond = newMax;
    }

    function setMinTargetUtilization(uint256 newVal) external {
```

```
        minimumTargetUtilization = newVal;
    }
    function setMaxTargetUtilization(uint256 newVal) external {
        maximumTargetUtilization = newVal;
    }
    function setStartInterestPerSecond(uint256 newVal) external {
        currentInterestPerSecond = newVal;
    }


    // Multiplicative issue / compound vs no compound
    // Reset via quick repay to re-do
    // Comparison of slopes

    // NOTE: `utilization` can be above 100%
    function updateIntestRate(uint256 utilization, uint256 elapsedTime)
        public
        returns (
            uint256 newRate
        )
    {
        uint256 fullUtilizationMinusMax = FULL_UTILIZATION - maximumTargetUtilization;
        // Update interest rate | /// @audit TODO: Charts + tests?
        if (utilization < minimumTargetUtilization) {

            uint256 underFactor = ((minimumTargetUtilization - utilization) *
                FACTOR_PRECISION) / minimumTargetUtilization;

            uint256 scale = interestElasticity +
                (underFactor * underFactor * elapsedTime);
            currentInterestPerSecond = uint64(
                (uint256(currentInterestPerSecond) * interestElasticity) /
                    scale
            );
            if (currentInterestPerSecond < minimumInterestPerSecond) {
                currentInterestPerSecond = minimumInterestPerSecond; // 0.25% APR minimum
            }
        } else if (utilization > maximumTargetUtilization) {
            uint256 overFactor = ((utilization - maximumTargetUtilization) *
                FACTOR_PRECISION) / fullUtilizationMinusMax;
            uint256 scale = interestElasticity +
                (overFactor * overFactor * elapsedTime);
            uint256 newInterestPerSecond = (uint256(
                currentInterestPerSecond
            ) * scale) / interestElasticity;
            if (newInterestPerSecond > maximumInterestPerSecond) {
                newInterestPerSecond = maximumInterestPerSecond; // 1000% APR maximum
            }

            currentInterestPerSecond = uint64(newInterestPerSecond);
        } /// @audit Stateful w/e to svg
    }
}

contract ExampleTest is Test {
    DemoInterestRate target;

    function setUp() public {
        target = new DemoInterestRate();
    }

    function testBasicOneDay() public {
```

```
        target.setMinInterest(158548960);
        target.setMaxInterest(317097920000);
        target.setMinTargetUtilization(3e17); // 10%
        target.setMaxTargetUtilization(5e17); // 80%

        target.setStartInterestPerSecond(158548960);

        console2.log("currentInterestPerSecond", target.currentInterestPerSecond());

        target.updateIntestRate(1e18,  1 days);
        console2.log("currentInterestPerSecond", target.currentInterestPerSecond());
    }

    function testBasicOneDayAlwaysAccrue() public {
        target.setMinInterest(158548960);
        target.setMaxInterest(317097920000);
        target.setMinTargetUtilization(3e17); // 10%
        target.setMaxTargetUtilization(5e17); // 80%

        target.setStartInterestPerSecond(158548960);

        console2.log("currentInterestPerSecond", target.currentInterestPerSecond());

        uint256 total;
        while(total < 1 days) {
            target.updateIntestRate(1e18,  1);
            total += 1;
        }
        console2.log("currentInterestPerSecond", target.currentInterestPerSecond());
    }

    // function testBasicCompare() public {
    //     target.setMinInterest(158548960);
    //     target.setMaxInterest(317097920000);
    //     target.setMinTargetUtilization(1e17); // 10%
    //     target.setMaxTargetUtilization(8e17); // 80%

    //     target.setStartInterestPerSecond(158548960);

    //     console2.log("currentInterestPerSecond", target.currentInterestPerSecond());

    //     target.updateIntestRate(1e18,  3 days);
    //     console2.log("currentInterestPerSecond", target.currentInterestPerSecond());
    // }

    function testReduceFromMax() public {
        target.setMinInterest(158548960);
        target.setMaxInterest(317097920000);
        target.setMinTargetUtilization(3e17); // 10%
        target.setMaxTargetUtilization(5e17); // 80%

        target.setStartInterestPerSecond(158548960);

        console2.log("currentInterestPerSecond", target.currentInterestPerSecond());

        target.updateIntestRate(7e17,  1 days);
        console2.log("currentInterestPerSecond", target.currentInterestPerSecond());
    }
    function testReduceFromMaxAFullDaily() public {
        target.setMinInterest(158548960);
        target.setMaxInterest(317097920000);
        target.setMinTargetUtilization(3e17); // 10%
```

```
        target.setMaxTargetUtilization(5e17); // 80%

        target.setStartInterestPerSecond(158548960);

        console2.log("currentInterestPerSecond", target.currentInterestPerSecond());

        uint256 total;
        while(total < 1 days) {
            target.updateIntestRate(7e17,  1);
            total += 1;
        }
        console2.log("currentInterestPerSecond", target.currentInterestPerSecond());
    }

    function testReduceFromMaxCompare() public {
        target.setMinInterest(158548960);
        target.setMaxInterest(317097920000);
        target.setMinTargetUtilization(3e17); // 10%
        target.setMaxTargetUtilization(5e17); // 80%

        target.setStartInterestPerSecond(158548960);

        console2.log("currentInterestPerSecond", target.currentInterestPerSecond());

        target.updateIntestRate(8e17,  1 days);
        console2.log("currentInterestPerSecond", target.currentInterestPerSecond());
    }
}
```

Which will yield

```
[PASS] testBasicOneDay() (gas: 128897)
Logs:
  currentInterestPerSecond 158548960
  currentInterestPerSecond 634195840

[PASS] testBasicOneDayAlwaysAccrue() (gas: 250412571)
Logs:
  currentInterestPerSecond 158548960
  currentInterestPerSecond 3184100010

[PASS] testReduceFromMax() (gas: 128918)
Logs:
  currentInterestPerSecond 158548960
  currentInterestPerSecond 234652460

[PASS] testReduceFromMaxAFullDaily() (gas: 250412616)
Logs:
  currentInterestPerSecond 158548960
  currentInterestPerSecond 256171112

[PASS] testReduceFromMaxCompare() (gas: 128864)
Logs:
  currentInterestPerSecond 158548960
  currentInterestPerSecond 329781836
```

As you can see, continously accruing has pretty dramatic impacts, in the case of a multi day scenario, this can cause a difference in interest rate that is up to 6 times higher when compounding more often.

Recommendation: Consider whether continous compounding should be factored-in into the formula, or whether

a bot / keeper should be used. Also consider a change to cause rates to increase exponentially when above a certain threshold, instead of always applying this compounding math to any utilization rate.

Tapioca: Acknowledged. This is the intended behavior. Interest accrues differently depending on utilization which is capped between MIN and MAX_UTILIZATION and the rate can increase up to max interest rate.

Spearbit: It is worth nothing that:

- Borrowers would be happy with this implementation as they would, in general, pay less than a continously compounded rate.
- Lenders will most likely consider accruing themselves as a means to ensure that compounded rates are enforced.

This may cause some gotchas to end users, specifically seeing a rate that changes by up to 5 / 6 times over a relatively short period of time, based on the frequency of accruals.

Overall, a nofix is acceptable, but we recommend extensively explaining this mechanism as it can be counter-intuitive to borrowers and lenders.

5.3.23 Under certain conditions, `toShares` and `toAmount` will return inconsistent results, which may be used to leak value

Severity: Medium Risk

Context: YieldBox

Description: This finding doesn't demonstrate a E2E impact due to:

- Timing constraints.
- Complexity of the setup and codebase required to demonstrate a higher E2E impact.

The finding shows that under specific pre-conditions, the result of `toShares` and `toAmount` from Yieldbox is not consistent, the main risk is that the rest of the mathematical assumptions may be broken due to this, and is notable because Yieldbox has undergone formal verification, and yet we demonstrate that if the first deposit is not done by the deployer, then some of the underlying invariants as to how Yieldbox Prices it's shares and assets are broken.

This doesn't show a specific risk for Yieldbox as Yieldbox consistently overprices amounts and underissues shares but can cause an issue to integrators (in this case BB and SGL), as they rely on `toAmount` and `toShare` as if they could be used interchangeably.

The specifics are the following; for any new asset that has no deposit, by depositing dust amounts we are able to:

- Have yieldbox issue a higher amount of shares when using `amount`.
- We always overpay shares (at best we receive them at fair value), when using `shares`.

Proof of Concept:

```
function testYBSimplestRoundingDemo() public {
    yb = new MockYieldBox();

    (uint256 amountFromShares, uint256 sharesFromShares) = yb.depositAsset(0, 1e18, 1e6, 1e6);
    (uint256 amountFromAmount, uint256 sharesFromAmount) = yb.depositAsset(amountFromShares, 0, 1e6,
↪   1e6);

    console2.log("amountFromShares", amountFromShares);
    console2.log("sharesFromShares", sharesFromShares);
    console2.log("amountFromAmount", amountFromAmount);
    console2.log("sharesFromAmount", sharesFromAmount);


    // After that, does the scenario change?

    (uint256 amountFromSharesAfterDeposit, uint256 sharesFromSharesAfterDeposit) = yb.depositAsset(0,
↪   1e18, amountFromShares + 1, sharesFromShares + 1);
    (uint256 amountFromAmountAfterDeposit, uint256 sharesFromAmountAfterDeposit) =
↪   yb.depositAsset(amountFromSharesAfterDeposit, 0, amountFromShares + 1, sharesFromShares + 1);
    console2.log("amountFromSharesAfterDeposit", amountFromSharesAfterDeposit);
    console2.log("sharesFromSharesAfterDeposit", sharesFromSharesAfterDeposit);
    console2.log("amountFromAmountAfterDeposit", amountFromAmountAfterDeposit);
    console2.log("sharesFromAmountAfterDeposit", sharesFromAmountAfterDeposit);
}
```

- Consistent Output (1e6 deposit):

```
[PASS] testYBSimplestRoundingDemo() (gas: 183501)
Logs:
  amountFromShares 9901000000000000
  sharesFromShares 1000000000000000000
  amountFromAmount 9901000000000000
  sharesFromAmount 1000000000000000000
  amountFromSharesAfterDeposit 9900999999009902
  sharesFromSharesAfterDeposit 1000000000000000000
  amountFromAmountAfterDeposit 9900999999009902
  sharesFromAmountAfterDeposit 1000000000000000000
```


- Inconsistent output (1 wei of initial deposit):

```
[PASS] testYBSimplestRoundingDemo() (gas: 183575)
Logs:
  amountFromShares 19999999801
  sharesFromShares 1000000000000000000
  amountFromAmount 19999999801
  sharesFromAmount 1000000000049999900
  amountFromSharesAfterDeposit 19999999801
  sharesFromSharesAfterDeposit 1000000000000000000
  amountFromAmountAfterDeposit 19999999801
  sharesFromAmountAfterDeposit 1000000000000000000
```

A complete testing repo is available here.

Elaboration:

The above demonstrates how, when dealing with small amounts of shares and assets, we are able to have the system over-estimate the amount of shares that will be issued given an amount, this may be usable as part of a more complex chain of depositing, borrowing, and rebases to SGL or BB.

Recommendation: Consider:

- If you can change the check in SGLCommon.sol#L254-L255:

```
if (_totalAsset.base < 1000) revert MinLimit();
```

To be based on the asset decimals, for example 10 ** decimals, or whether Yieldbox can always be seeded with an amount that would be above 1e8 as to mitigate this attack under many circumstances.

This would be possible but would restrict the protocol to be usable for tokens with at least 8 decimals (exception being USDC which would require $100 of original deposit) and will eliminate additional rounding risks at the SGL level.

- Always performing an initial deposit on Yieldbox as to ensure that no token share is rebaseable.

Tapioca: Acknowledged.

Spearbit: Acknowledged.

5.3.24 `oTAP.participate()` will always revert if `msg.sender` is approved but not owner

Severity: Medium Risk

Context: TapiocaOptionBroker.sol#L295-L299, Code4rena Issue#349

Description: As confirmed in Code4rena Issue#349, `tOLP.transferFrom(msg.sender, address(this), _tOLPTokenID)`; will always revert when `msg.sender` is approved but not the owner of the token.

Impact: Approved address/operator of `oTAP.participate()` will always revert.

Likelihood: Medium (requires an approved call) + Impact: Medium = Severity: Medium.

Recommendation: As recommended in Code4rena Issue#349, transfer from the owner instead of `msg.sender`:

```
address owner = tOLP.ownerOf(_tOLPTokenID);
tOLP.transferFrom(owner, address(this), _tOLPTokenID);
```

Tapioca: Created PR 136.

5.3.25 Users with a smart-contract wallet address may have funds drained from an attacker who controls that address on another chain

Severity: Medium Risk

Context: BaseTOFTGenericModule.sol#L201-L293

Description: `triggerSendFrom()` allows a user on a source chain to trigger a `sendFrom()` from a destination chain, which initiates a transfer of TOFT's from the user's address on the destination chain to the same address on the source chain. This flow and others in the protocol assume the same ownership of an address across all chains. While this is true for Externally-Owned-Accounts (EOAs) which rely on private-keys, this is not the case for Smart-Contract-Accounts (SCAs), which are used by smart contract wallets such as Safe. In SCAs, the ownership and other properties of the account are specific to the code on deployed chains and therefore can be different across chains. This aspect has already been exploited in the past for Safe wallets across Ethereum and Optimism.

The protocol acknowledges this threat and will advise their users to not use SCAs but only EOAs for interacting with the protocol. However, smart contract wallets are increasingly popular for security or account abstraction reasons (e.g. Safe wallets already manage billions of dollars across individual users and DAOs), and will only increase with ongoing efforts such as ERC4337.

Impact: Any user mistakenly interacting with the protocol using a smart-contract wallet (e.g. Safe) address on one chain, say Chain-1, may have funds drained by an attacker who controls that address on another chain, say Chain-2, by the attacker triggering a `triggerSendFrom()` from Chain-2 to the user's address on Chain-1 (which is the same as the attacker controlled address). Given the assumption of same ownership of addresses across Chain-1 and Chain-2, the protocol will send user's TOFT on Chain-1 to the attacker on Chain-2.

Likelihood: Low (depends on users ignoring protocol UI warnings and attacker controlling the same address on another chain) + Impact: High = Severity: Medium.

Recommendation:

1. Consider removing `triggerSendFrom()` entirely and instead requiring users to push funds cross-chain via `triggerSendFromWithParams()` instead of pulling from another chain.
2. Reevaluate all cross-chain protocol interactions such as `triggerSendFrom()` and refund flows, which assume same ownership of addresses across supported chains and consider re-architecting them to avoid this assumption.

Tapioca: Acknowledged. We won't allow multisigs

Spearbit: It is unclear how this will be enforced.

5.3.26 `sendToYBAndBorrow()` may fail on destination chain due to ignored `extraGasLimit` consideration on source chain

Severity: Medium Risk

Context: BaseTOFTMarketModule.sol#L123, ICommonData.sol#L15, BaseTOFTMarketModule.sol#L158, LzApp.sol#L56-L61, BaseTOFTStrategyModule.sol#L84 and similar USDO modules.

Description: `sendToYBAndBorrow()` expects the user to provide `ICommonData.ISendOptions calldata options` as a argument which contains the `extraGasLimit` field besides the `zroPaymentAddress` field. While `options.zroPaymentAddress` is used in `_lzSend()`, `options.extraGasLimit` is ignored in the `_checkAdapterParams()` where `NO_EXTRA_GAS` is used instead.

In the `_checkGasLimit()` validation of `_checkAdapterParams()`:

```
function _checkGasLimit(uint16 _dstChainId, uint16 _type, bytes memory _adapterParams, uint _extraGas)
↪  internal view virtual {
    uint providedGasLimit = _getGasLimit(_adapterParams);
    uint minGasLimit = minDstGasLookup[_dstChainId][_type] + _extraGas;
    require(minGasLimit > 0, "LzApp: minGasLimit not set");
    require(providedGasLimit >= minGasLimit, "LzApp: gas limit is too low");
}
```

Using `NO_EXTRA_GAS` instead of `options.extraGasLimit` for `_extraGas` could incorrectly satisfy the `providedGasLimit >= minGasLimit` check while it may have failed with `extraGasLimit` consideration.

Similar use of `options` parameter in `BaseTOFTStrategyModule.sendToStrategy()` uses `options.extraGasLimit` in the call to `_checkAdapterParams()` indicating that this field needs to be considered while accepting the `options` argument from the user.

Impact: `sendToYBAndBorrow()` may fail on destination chain.

Likelihood: Medium (depends on user providing a non-zero `options.extraGasLimit` value) + Impact: Medium = Severity: Medium.

Recommendation: Use `options.extraGasLimit` instead of `NO_EXTRA_GAS` in the `_checkAdapterParams()` validation for `sendToYBAndBorrow()`. Evaluate all TOFT and USDO flows for this issue.

Tapioca: Created PRs 135 and 306.

5.3.27 Unchecked revert message lengths may lead to protocol-wide DoS

Severity: Medium Risk

Context: BaseUSDOStorage.sol#L85-L96, BaseTapOFT.sol#L170-L171, BaseTapOFT.sol#L315-L316, BaseTapOFT.sol#L428-L429

Description: As described in detail in C4-27, an attacker can grief the protocol by reverting with long messages from their controlled contracts such that it consumes all provided gas while processing them to force OOG exception at LayerZero Endpoints of Tapioca and cause all message passing to be blocked thereafter.

While this was reported earlier in C4-27 and partially fixed in Penrose.sol#L499, Market.sol#L405, BaseTOFTStorage.sol#L109, there were other places as referenced above where this mitigation is missing.

Impact: No new cross-chain messages can be relayed in Tapioca if attacker griefs via these vectors.

Likelihood: Medium + Impact: Medium = Severity: Medium.

Recommendation: Check the length of revert messages before processing them.

Tapioca: Created PRs 304 and 135.

5.3.28 Failure of `triggerSendFromWithParams()` on the destination chain will lead to loss of user TOFT on source chain

Severity: Medium Risk

Context: BaseTOFTGenericModule.sol#L67, BaseTOFTGenericModule.sol#L89-L147

Description: To handle failures of cross-chain transactions, the protocol follows a recovery pattern where if any protocol transaction debits TOFT on the source chain then the corresponding destination chain counterpart detects failures/reverts by delegating the transaction logic to a delegatecall, detecting failure and then refunding/crediting the user address on the destination chain with the same TOFT amount as was debited on the source chain. It also stores the failed message details for future retries.

However, `triggerSendFromWithParams()` which is used to simply send TOFT from the source to destination chain, with an optional unwrap, is missing this recovery pattern. The corresponding `executSendFromWithParams()` on the destination chain simply credits to the specified `toAddress` and optionally unwraps it to the underlying ERC20/ETH. The `unwrap()` or `toAddress.call()` could potentially revert due to user error e.g. providing a `toAddress` incapable of receiving ETH.

Impact: User TOFT tokens are debited on the source chain but an equivalent amount is not credited on the destination chain due to any reverting failure in `executSendFromWithParams()`, which leads to a loss of user funds.

Likelihood: Low + Impact: High = Severity: Medium

Recommendation: Implement the failure recovery pattern in `executSendFromWithParams()` for the destination chain.

5.3.29 `_accrue()` may overflow when `extraAmount` is downcast or added to `_totalBorrow.elastic`

Severity: Medium Risk

Context: BBCommon.sol#L89-L95

Description: if appropriate limits are not set on the `maxDebtRate` and `totalBorrowCap` state variables then downcasting `extraAmount` from `uint256` to `uint128` runs the risk of an overflow after extended periods of disuse. When adding `extraAmount` to `_totalBorrow.elastic` there is also the possibility of throwing an overflow error until `extraAmount` is large enough that it overflows in the downcast instead.

```
function _accrue() internal override {
    // ...
    uint256 extraAmount = 0;
    extraAmount =
        (uint256(_totalBorrow.elastic) *
            _accrueInfo.debtRate *
            elapsedTime) /
        1e18;
    _totalBorrow.elastic += uint128(extraAmount);
    //...
}
```

Recommendation: Define how long the protocol is expected to go without any functions that call `_accrue` to be interacted with, and setting the `maxDebtRate` and `totalBorrowCap` accordingly.

Cap `extraAmount` to `type(uint128).max - totalBorrowCap` to avoid halting of the protocol after extended periods of disuse.

Tapioca: Fixed in PR 303.

5.3.30 Incorrect refund address causes loss of gas refund to users

Severity: Medium Risk

Context: BaseTOFTStrategyModule.sol#L148, BaseTOFTGenericModule.sol#L80, BaseTOFTGenericModule.sol#L166, BaseTOFTGenericModule.sol#L236, USDOGenericModule.sol#L38, USDOGenericModule.sol#L106

Description: Cross-chain calls require user to provide for gas on the destination chain. Any excess gas supplied is expected to be refunded back to the user on their provided address. Given the unpredictability of accurately estimating cross-chain gas usage, it is likely that users may overpay for successful execution of cross-chain calls and expect refunds for excess gas provided.

However, `BaseTOFTStrategyModule.retrieveFromStrategy()`, `BaseTOFTGenericModule.triggerSendFromWithParams`, `BaseTOFTGenericModule.triggerSendFrom`, `USDOGenericModule.triggerApproveOrRevoke()` and `USDOGenericModule.triggerSendFrom()` incorrectly set the refund address to the delegated caller `msg.sender`. The refund address should either be `LzCallParams.refundAddress` if present, or the user-provided `from` address in their `_lzSend()` calls.

Impact: Any excess gas is refunded to the delegated caller `msg.sender` (i.e. operator) but not to the user-provided refund address, leading to loss of user gas refunds.

Likelihood: Medium + Impact: Medium = Severity: Medium.

This is similar to C4-1174.

Recommendation: Consider using user-provided address for gas refunds.

Tapioca: Created PRs 302, 132 and 163.

5.3.31 `SGLLeverage.buyCollateral` allows any caller to `withdraw` `supplyShare` as long as `collateralShare` rounds down to 0

Severity: Medium Risk

Context: SGLLeverage.sol#L23-L72

Description: `SGLLeverage.buyCollateral` allows to specify a `borrowAmount` and a `supplyAmount`. The invariant protecting against arbitrary callers is: `_allowedBorrow(from, collateralShare);` will have a non-zero `collateralShare` on any call, leading to the check for allowance on any amount.

However, `supplyShares` are swapped from `asset` to `collateral`, meaning that as long as we can get the `collateralShare` to result in a 0 amount, we can actually move any amount that was approved by `from`. The attack would be as follows:

- Victim deposits asset into yieldbox.
- Victim approves SGL for trading.
- Attacker calls `buyCollateral` with `borrowAmount = 0` and `supplyAmount != 0`.
- Attacker will imbalance the pool used by the swapper, as well as use as much slippage as possible to cause `amountOut` to be an extremely low value.
- `collateralShare` is computed via a `roundDown` meaning it can return a 0 amount.

The specifics on the attack are reliant on the specific token, as well as the total supply on Yieldbox. Based on the asset token, and the state of yieldbox, the attack may be extremely easy to perform, which would warrant a higher severity (critical in certain cases). For the sake of time we demonstrate the pre-condition to the exploit, which is that anybody can perform the call as long as the total amount will result in a 0.

```
it.only('Can I buy coll on behalf of someone else?', async () => {
    const {
        deployer,
        mockSwapper,
        weth,
        usd0,
        wethId,
        yieldBox,
        wethBigBangMarket,
        bar,
        eoa1,
        timeTravel,
        cluster,
    } = await loadFixture(setUp);

    await cluster.updateContract(
        await hre.getChainId(),
        mockSwapper.address,
        true,
    );

    await cluster.updateContract(
        await hre.getChainId(),
        wethBigBangMarket.address,
        true,
    );
    expect(
        await wethBigBangMarket.userBorrowPart(deployer.address),
    ).to.equal(E(10_000).div(10_000));
    const ybBalance = await yieldBox.balanceOf(
        deployer.address,
        await bar.usdoAssetId(),
    );
```

```
    expect(ybBalance.eq(E(1).mul(1e8))).to.be.true;

    //prefund swapper with some USDO
    await prefundSwapper(
        mockSwapper.address,
        yieldBox,
        wethBigBangMarket,
        weth,
        usdO,
        eoa1,
        wethId,
        await wethBigBangMarket.assetId(),
        timeTravel,
        false,
    );

    // //prefund swapper with some WETH
    // await weth.freeMint(E(10));
    // await weth.transfer(mockSwapper.address, E(10));

    const collateralBefore =
        await wethBigBangMarket.userCollateralShare(deployer.address);
    const borrowBefore = await wethBigBangMarket.userBorrowPart(
        deployer.address,
    );
    const ybBalanceOfDeployerAssetBefore = await yieldBox.balanceOf(
        deployer.address,
        await wethBigBangMarket.assetId(),
    );
    const encoder = new ethers.utils.AbiCoder();
    const leverageData = encoder.encode(
        ['uint256', 'bytes'],
        [0, []], // NOTE: Need zero so we get below 1 share, else this will not work
    );
    // Buy more collateral | Random Caller can perform the operation
    await wethBigBangMarket.connect(eoa1).buyCollateral(
        deployer.address,
        0, // One ETH; in amount
        0, // No additional payment
        leverageData,
    );
}),
```

Recommendation: Enforce a check that `collateralShare` are non-zero as to protect the invariant and avoid the attack entirely (see SGLLeverage.sol#L65-L72):

```
uint256 collateralShare = yieldBox.toShare(
    collateralId,
    amountOut,
    false /// @audit This can round down to zero
);
require(collateralShare!=0, "Non-zero"); /// @audit Enforce non-zero to avoid arbitrary msg.sender
_allowedBorrow(from, collateralShare);
_addCollateral(from, from, false, 0, collateralShare, false);
```

Tapioca: PR created (see PR 299).

Spearbit: Fixed by adding a customError on 0 `collateralShare`. Verified.

5.3.32 `Yieldbox permit` signatures do not specify a revoke or an approval, allowing the `msg.sender` to decide

Severity: Medium Risk

Context: YieldBoxPermit.sol#L61-L80

Description: YieldBox permit implements `permitAll revokeAll` as well as `permit` and `revoke`. Signatures for these are distinguished by the `TYPEHASH` they use (which separates the single `permit | revoke` vs. the multi `permitAll | revokeAll`).

However, neither of these schemes distinguishes between what is an approval and a revoke of said approval. This allows the recipient of the signatures to decide whether to use the signature to approve themselves, meaning that a `revoke` signature can be use to `permit` an operation.

This is inconsistent with how EIP 2612 is implemented (which specifies the amount when approving ERC20s).

Arguably the implementation follows this Stagnant EIP, meaning that some people may argue the implementation is correct, I would argue the opposite as this implementation makes it so that one signature, can have 2 meanings that are diametrically opposite.

Recommendation: Consider flagging this risk to end users, that they will have to revoke approvals by performing a transaction and never via a permit. More specifically:

- A permit could be used to approve an operator.

In case of wanting to revoke said approval, the end user will have to:

- Increase their nonce, as to invalidate any non-used permits.
- Revoke the approval via `revoke` or `revokeAll`.

Tapioca: Created PR 1.

5.3.33 `USDO` cross chain functionality may be denied based on LayerZero Settings

Severity: Medium Risk

Context: USDOCommon.sol#L70-L77

Preamble: As flagged in the review, USDO can allow for arbitrary calls. In many cases, said arbitrary calls would just cause reverts, which would be captured and would be safe under most circumnstances.

The following report shows hypothetical scenarios in which said reverts could be made to cause a revert to the LayerZero Non blocking app, in spite of it using a try-catch.

Description: These attacks rely on LayerZero enforcing ordered nonces. By causing reverts after the try/catch, we are able to prevent the LayerZero relayer from broadcasting the next message. We are able to cause reverts by using exponential memory expansion costs.

Due to the enforcing of ordered nonces, even a non-blocking app can be made to be blocking, as long as we can find a way to crash the try / catch mechanism.

Proof of concept: We demonstrate the risk by passing in 32kb of data in the payload, because of the hashing and the emissions of the payload happens after the non-blocking call, memory expansion costs can be used to cause a revert that cannot be caught:

```
// SPDX-License Identifier: MIT

pragma solidity 0.8.17;

import "forge-std/Test.sol";
import "forge-std/console2.sol";


contract GasConsumer {
```

```
    event FailedMessage(bytes _payload);
    //     event MessageFailed(uint16 _srcChainId, bytes _srcAddress, uint64 _nonce, bytes _payload,
↪   bytes _reason);

    function handleTheCall(bytes memory x) external {

        // Simulates mload from Safe Call
        assembly {
            pop(
                add(x, 0x20)
            )
            pop(
                mload(x)
            )
        }

        // NOTE: Here we would burn the remaining 63/64 after the mem-expansio
        // Since mem expansion costs are not linear, we should be able to still DOS because the
↪   expansion will continue
        // While ideally we would burn gas related to memory after the call,
        // This is fine as we have 2 more entry points to cause mem-expansion become supralinear

        bytes32 hashed = keccak256(x);

        emit FailedMessage(x); // TODO: See how much it consts in relation
    }
}

contract ExampleTest is Test {
    GasConsumer consumer;

    function setUp() public {
        consumer = new GasConsumer();
    }

    function testSomeGas() public {
        uint256 startGas = gasleft();
        consumer.handleTheCall{gas: 30_000_000}(new bytes(32 * 1000)); // Around 32 * 1000 = 32kbs
        uint256 endGas = gasleft(); // 512k gas is gas cost to pass the calldata assuming non-zero
        console2.log("delta", startGas - endGas);
    }
}
```

Some math suggests that this can be possible by passing in duplicate approval targets (as to sidestep allowed targets). Each approval has 7 fields:

```
approval.owner,
approval.spender,
approval.value,
approval.deadline,
approval.v,
approval.r,
approval.s
```

Bytes abi.encoding is structured in the following way:

- Offset.
- Length.
- Data.

If the Data is seven 32bytes words, we can add 2 words for Offset and Length, giving us 9 words. Since we have

shown we can crash any receiver with 32 * 1k bytes, we just need 1k words of data. One thousand words / 9 words per approval gives us 111.11 (repeating) approvals. This gives us an upper bound for the attack.

The scenarios in which we can call an arbitrary target, the gas requirements will be way lowered, specifically: The gas requirement will be 1/64 of the max gas per block which will be kept by the `excessivelySafeCall`, in the case of a 30 MLN max block, that value would be: `468750`.

Meaning that the memory expansion cost simply has to cover that amount for any call to revert.

Recommendation:

- Cap the Payload length to a reasonable amount, 10 or so approvals should be plenty for any benign user.
- Setup the whole system E2E, then do a security review of the setup system, it is impossible with the given scope to devise any additional specific attack because we don't have access to the exact settings that you will use with LayerZero.
- Consider the risks of CrossChain operations being severely delayed and recommend end users to separate bridging from more time sensitive operations.

To limit the payload length, you can use:

```
function setPayloadSizeLimit(uint16 _dstChainId, uint _size) external onlyOwner {
    payloadSizeLimitLookup[_dstChainId] = _size;
}
```

And set to a size that is less than 1 thousand bytes.

Tapioca: Fixed by capping payload length to 1k bytes.

Spearbit: The fix should prevent the attack in most circumnstances. It's highly recommended that E2E tests are performed on the final setup. Some of the E2E recommendations have been written in "E2E LayerZero Testing Checklist".

A security review of the finalized system is recommended as any small change could bring back potential attacks to the nonBlockingApp.

5.3.34 `Singularity` rounding math can favour the caller and may lead to economic exploits

Severity: Medium Risk

Context: SGLCommon.sol#L220-L221

Preamble: The following is a review of the rounding decisions throughout a simplified flow for Singularity

The finding is sent early to help mitigate these risks, as I believe that if we spent sufficient time, we would be able to come up with an economic attack that is related to rounding. Due to time constraints as well as the need to cover other code, I'm sending this early for the team to review, as we can then re-prioritize this issue based on the progress of the review in other areas of the codebase

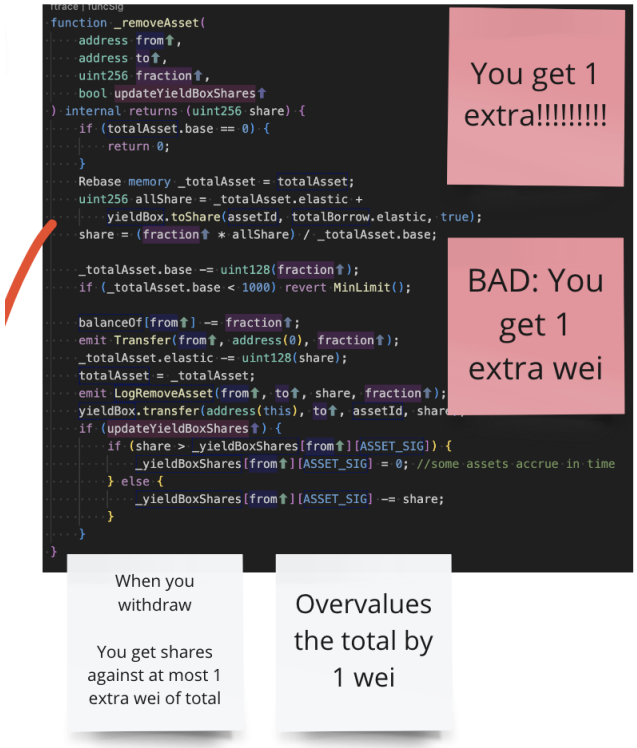
Description:

1. When `_addAsset` is called the depositor will lose at most 1 wei of the credited amount.

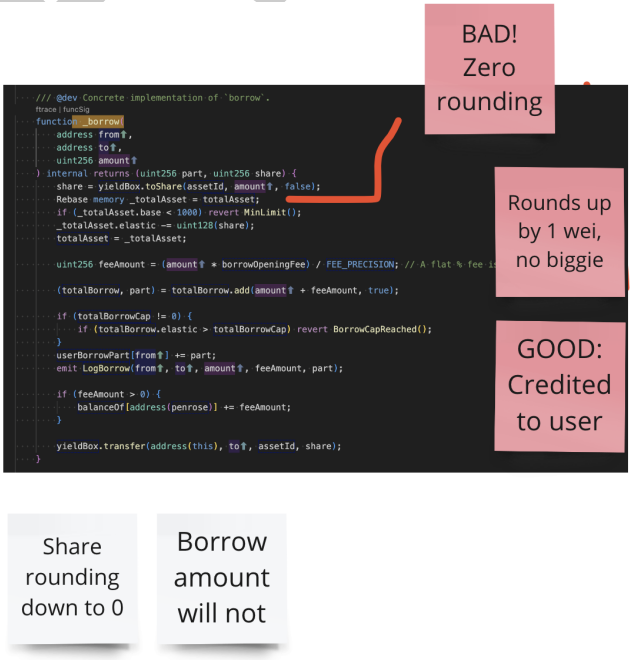
This ensures that no "ghost share" can be minted, which is positive.

2. When `removeAsset` is called, the withdrawer will receive a pro-rata distribution based on a `roundUp` of shares that have been lent.

   This can reduce the value of the basket of Asset + Debt, effectively giving away some value to the caller, leaking value (and potentially leading to more issues).



3. When `_borrow` is called, the amount of `shares` that are withdrawn is rounded down and this rounded down value is subtracted from `_totalAsset.elastic`.



This effectively allows the risk mentioned above (2), as we are reducing potentially a 0 wei share of debt, but we are "paying out" 1 extra wei of said debt. In constract, the rounding of the `totalDebt` is positive as the caller is paying that extra wei of amount which should have negligible impacts to them and to the system

4) In the case of `_repay` the `sub(part, true` is forgiving at most 1 wei of debt.

This should be changed as it may lead to slight undercollateralization, however, the impact seems to be limited to dust amounts.

Recommendation: Review all rounding decisions and document them. Generally speaking:

- Rounding when computing amounts to shares should be up, requiring more value before issuing shares.
- Rounding for withdrawals should be down as to release less amount to the caller rather than more.

Due to the high complexity of the codebase, invariant testing may be necessary to iron out edge cases at that level, as a single rounding in favour of the caller may be exploited to leak value or break the collateralization of the system via overborrowing.

5.3.35 `SGLLendingCommon._borrow feeAmount` is not subtracted from `totalBorrow.elastic` which causes the SGL shares to rebase incorrectly

Severity: Medium Risk

Context: SGLLendingCommon.sol#L67-L91

Description: `SGLLendingCommon._borrow` charges a borrowing fee in the following way (see SGLLendingCommon.sol#L67-L91):

```
function _borrow(
    address from,
    address to,
    uint256 amount
) internal returns (uint256 part, uint256 share) {
    share = yieldBox.toShare(assetId, amount, false);
    Rebase memory _totalAsset = totalAsset;
    if (_totalAsset.base < 1000) revert MinLimit();
    _totalAsset.elastic -= uint128(share);
    totalAsset = _totalAsset;

    uint256 feeAmount = (amount * borrowOpeningFee) / FEE_PRECISION; // A flat % fee is charged for any
↪   borrow

    (totalBorrow, part) = totalBorrow.add(amount + feeAmount, true); /// @audit Something is off here

    if (totalBorrowCap != 0) {
        if (totalBorrow.elastic > totalBorrowCap) revert BorrowCapReached();
    }
    userBorrowPart[from] += part;
    emit LogBorrow(from, to, amount, feeAmount, part);

    if (feeAmount > 0) {
        balanceOf[address(penrose)] += feeAmount;
    }
```

This fee is added to the `totalBorrow` which is used to determine the value of a new asset deposit, via the following formula (see SGLCommon.sol#L219-L223):

```
uint256 allShare = _totalAsset.elastic +
    yieldBox.toShare(assetId, totalBorrow.elastic, true);
fraction = allShare == 0
    ? share
    : (share * _totalAsset.base) / allShare;
```


After an asset has been borrowed, this formula will credit the extra debt to the basket, to new depositors, instead of it being paid by them. In contrast `_accrue` fee amounts are computed by increasing `base` which effectively dilutes the value of each deposits according to the interest to be paid (see SGLCommon.sol#L113-L116):

```
uint256 feeAmount = (extraAmount * protocolFee) / FEE_PRECISION; // % of interest paid goes to fee
feeFraction = (feeAmount * _totalBorrow.base) / fullAssetAmount;
_accrueInfo.feesEarnedFraction += uint128(feeFraction);
_totalAsset.base = _totalAsset.base + uint128(feeFraction);
```

The math inconsistency can be demonstrated via this simplified proof of concept:

We compare the `allShare` we will have if we take a loan, and we demonstrate that due to the math, once all loans are repaid, the `allShare` will cause Singularity to allow asset depositors to "steal" the protocol fee, since it is not deducted by `totalDebt.base` .

Proof of concept:

```
[PASS] testBasicDepositCompareBorrowMath() (gas: 844314)
Logs:

  Counter 0

  assetElastic 1000000000000000000
  assetBase 1000000000000000000
  borrowElastic 0
  borrowBase 0
```

```
  userBorrowPart(FROM) 0
  deposited(FROM) 1000000000000000000
  Asset Implied PPFS 1000000000000000000
  allShare 1000000000000000000

  Counter 1

  assetElastic 500000000000000000
  assetBase 1000000000000000000
  borrowElastic 505000000000000000
  borrowBase 505000000000000000
  userBorrowPart(FROM) 505000000000000000
  deposited(FROM) 1000000000000000000
  Asset Implied PPFS 500000000000000000
  allShare 1005000000000000000

  Counter 2

  assetElastic 1500000000000000000
  assetBase 1995024875621890547
  borrowElastic 505000000000000000
  borrowBase 505000000000000000
  userBorrowPart(FROM) 505000000000000000
  deposited(FROM) 1995024875621890547
  Asset Implied PPFS 751870324189526184
  allShare 2005000000000000000

[PASS] testBasicDepositCompareNoBorrowMath() (gas: 786895)
Logs:

  Counter 0

  assetElastic 1000000000000000000
  assetBase 1000000000000000000
  borrowElastic 0
  borrowBase 0
  userBorrowPart(FROM) 0
  deposited(FROM) 1000000000000000000
  Asset Implied PPFS 1000000000000000000
  allShare 1000000000000000000

  Counter 1

  assetElastic 2000000000000000000
  assetBase 2000000000000000000
  borrowElastic 0
  borrowBase 0
  userBorrowPart(FROM) 0
  deposited(FROM) 2000000000000000000
  Asset Implied PPFS 1000000000000000000
  allShare 2000000000000000000
```


Code:

```
// SPDX-License Identifier: MIT

pragma solidity 0.8.17;

import "forge-std/Test.sol";
import "forge-std/console2.sol";
import "src/MockDepositRepay.sol";
```

```
contract DepositBorrow is Test {
    MockDepositRepay mockSgl;

    address FROM = address(0xb4d455);

    uint256 counter;

    function _logState() internal {
        console2.log("");
        console2.log("Counter", counter++);
        console2.log("");

        (uint128 assetElastic, uint128 assetBase) = mockSgl.totalAsset();
        console2.log("assetElastic", assetElastic);
        console2.log("assetBase", assetBase);

        (uint128 borrowElastic, uint128 borrowBase) = mockSgl.totalBorrow();

        console2.log("borrowElastic", borrowElastic);
        console2.log("borrowBase", borrowBase);

        console2.log("userBorrowPart(FROM)", mockSgl.userBorrowPart(FROM));
        console2.log("deposited(FROM)", mockSgl.deposited(FROM));

        console2.log("Asset Implied PPFS", assetElastic * 1e18 / assetBase);



        uint256 allShare = assetElastic +
            mockSgl.yieldBox_toShare(borrowElastic);
        console2.log("allShare", allShare);
    }

    function testBasicDepositCompareNoBorrowMath() public {
        mockSgl = new MockDepositRepay();

        uint256 AMT = 1e18;

        // Deposit 1e18
        mockSgl._addAsset(FROM, FROM, false, AMT); // 1e18 to 1e18

        _logState();

        // Deposit again
        mockSgl._addAsset(FROM, FROM, false, AMT);
        _logState();

        // See what happens


    }
    function testBasicDepositCompareBorrowMath() public {
        mockSgl = new MockDepositRepay();

        uint256 AMT = 1e18;

        // Deposit 1e18
        mockSgl._addAsset(FROM, FROM, false, AMT); // 1e18 to 1e18

        _logState();

        // Borrow Half
```

```
        mockSgl._borrow(FROM, FROM, AMT / 2);
        _logState();

        // Deposit again
        mockSgl._addAsset(FROM, FROM, false, AMT);
        _logState();

        // See what happens
    }

}
```

Note: Full proof of concept repository is available here.

Recommendation: Currently, the best recommendation would be to not deposit the fees, and credit them directly as an asset or as Yieldbox shares. This will simplify the accounting, as many other possible fixes will end up inflating the value of `fractions`.

5.3.36 Unspecified allowance spend for delegation checks may cause loss of allowance for approved operators

Severity: Medium Risk

Context: BaseTOFTMarketModule.sol#L59, USDOMarketModule.sol#L48-L52, USDOMarketModule.sol#L61-L65

Description: Various USDO/TOFT cross-chain operations allow users to delegate them to approved operators. However, instead of only enforcing an allowance check when `msg.sender != user` to limit execution to approved operators, the operations also consume/spend the approved allowance even when no funds are being spent from the user. This is unnecessary unless it is used to limit the number of such delegated operations, which is not specified.

These check+consume allowances appear to have been added as a mitigation to C4-1032 and related issues from the previous security review. While the allowance checks may be sufficient to enforce authorized delegated operations, the allowance spend may not be necessary.

Impact: If `msg.sender` is approved to have the required allowance then it will be unnecessarily consumed as part of these authorization checks for delegation. This scenario will either require the user to thereafter perform the previously delegated operations themselves or cause loss of approved allowance for the delegated operator.

Likelihood: Medium + Impact: Medium = Severity: Medium.

Recommendation: Evaluate allowance consumption for delegated operations and either specify/document expected behavior or consider removal.

Tapioca: Acknowledged. We'll leave it as it is for now.

Spearbit: Acknowledged.

5.3.37 Possible DOS on repayment in case of credit delegation

Severity: Medium Risk

Context: BBLendingCommon.sol#L57, BBLendingCommon.sol#L117

Description: This issue is dependent on issue "User can manipulate the borrow/repay mechanism to cause loss of `openingFee` for the protocol" where in the case of credit delegation methodology of using different `from`/`to` addresses, as described in the proof of concept of the aforementioned issue.

As seen the user's `openingFee` gets accumulated over time and is never cleared. Hence in case of an actual direct borrow for this user the repayment would revert if the collected `openingFee` is substantial enough to be greater than the borrow `part` they intend to repay, because of the following condition: `if (openingFee >= part) revert RepayAmountNotValid();`

Likelihood: Low + Impact: High = Severity: Medium.

Recommendation: Consider maintaining the same address while accounting for `openingFees` and `userBorrowPart`.

5.3.38 Missing whitelist checks on user-provided addresses allow arbitrary external calls in options destination modules

Severity: Medium Risk

Context: BaseTOFTOptionsDestinationModule.sol#L156-L173, USDOOptionsDestinationModule.sol#L137-L155

Description: While option destination modules apply whitelist check on `tapSendData.tapOftAddress`, they are missing similar whitelist checks on user-provided `optionsData.paymentToken` and `optionsData.target` addresses.

Impact: While it is not immediately clear how this may be exploitable, this allows users to make arbitrary external calls in the context of leverage destination modules, which at the very least may be used for griefing but extremely risky in the worst case where underlying assets may be drained.

Likelihood: High + Impact: Low (Assuming griefing) = Severity: Medium.

Recommendation: Apply whitelist checks for all user-provided addresses on both source and destination chains. Perform input validation on all user-provided data, especially those that may be/contain addresses and asset identifiers.

Tapioca: Fixed in commit 687381484.

5.3.39 Target debt size is controlled only after allowance reduction, and extra allowance is removed on each repay with amount bigger than actual debt

Severity: Medium Risk

Context: BBBorrow.sol#L86-L88, BBLendingCommon.sol#L111-L113, SGLBorrow.sol#L83-L85, SGLLendingCommon.sol#L102-L104, MarketERC20.sol#L79-L89

Description: When user supplied `part` is so that `part > userBorrowPart[to]`, the allowance for the full `part` will be written off, as only `userBorrowPart[to]` be then used. I.e. the allowance for the difference will be just lost for the caller.

This happens on repay only as it limits the amount:

- BBBorrow.sol#L86-L88:

```
_allowedBorrow(from, allowanceShare);

amount = _repay(from, to, part);
```

- SGLBorrow.sol#L83-L85

```
_allowedBorrow(from, allowanceShare);

amount = _repay(from, to, skim, part);
```

`_allowedBorrow()` will write down extra allowance each time it be called with extra `part`:

- MarketERC20.sol#L79-L89

```
function _allowedBorrow(address from, uint share) internal {
    if (from != msg.sender) {
        require(
            allowanceBorrow[from][msg.sender] >= share,
            "Market: not approved"
        );
        if (allowanceBorrow[from][msg.sender] != type(uint256).max) {
            // see the line below
            allowanceBorrow[from][msg.sender] -= share;
        }
    }
}
```

As `part` is being controlled to not exceed the total debt of `to` only in `_repay()`, after allowance was reduced already:

- BBLendingCommon.sol#L111-L113

```
if (part > userBorrowPart[to]) {
    part = userBorrowPart[to];
}
```

- SGLLendingCommon.sol#L102-L104

```
if (part > userBorrowPart[to]) {
    part = userBorrowPart[to];
}
```

Impact: when user supplied `part` is so that `part > userBorrowPart[to]`, the allowance for the full `part` will be written off, while only `userBorrowPart[to]` be then used. I.e. the difference will be lost for the caller.

Per high likelihood and low impact setting severity to be medium.

Recommendation: Consider spending allowance only on the final amount to be used.

5.3.40 Missing whitelist checks on user-provided addresses allow arbitrary external calls in market destination modules

Severity: Medium Risk

Context: BaseTOFTMarketModule.sol#L116-L171, BaseTOFTMarketDestinationModule.sol#L46-L99, BaseTOFTMarketDestinationModule.sol#L162-L173, USDOMarketModule.sol#L35-L36, USDOMarketDestinationModule.sol#L207-L209

Description: `BaseTOFTMarketModule` and `BaseTOFTMarketDestinationModule` modules are missing whitelist checks on user-provided `borrowParams.market` and `borrowParams.marketHelper`.

Similarly, `USDOMarketModule` and `USDOMarketDestinationModule` are missing whitelist checks on user-provided addresses in `externalData` and `removeAndRepayData`. While some of these addresses are checked further downstream within `Magnetar.exitPositionAndRemoveCollateral()`, `externalData.magnetar` is used as the address for this call without any validation.

Impact: While it is not immediately clear how this may be exploitable, this allows users to make arbitrary external calls in the context of market destination modules, which at the very least may be used for griefing but extremely risky in the worst case where underlying assets may be drained.

Likelihood: High + Impact: Low (Assuming griefing) = Severity: Medium.

Recommendation: Check whitelist status of `borrowParams.market`, `borrowParams.marketHelper` and all addresses within `externalData` and `removeAndRepayData` on both source and destination chains. Perform input validation on all user-provided data, especially those that may be/contain addresses and asset identifiers.

Tapioca: Fixed in PR 121.

5.3.41 Delegated cross-chain TOFT retrieval may unnecessarily consume allowance

Severity: Medium Risk

Context: BaseTOFTStrategyModule.sol#L115-L119

Description: `BaseTOFTStrategyModule.retrieveFromStrategy()` attempts to extract TOFT deposited to a cross-chain strategy on another layer previously via `BaseTOFTStrategyModule.sendToStrategy()`. It allows a user to delegate the TOFT retrieval to another address similar to other operations. However, it enforces an allowance check and consumption on the `msg.sender` when `_from != msg.sender`. Given that this retrieval operation is receiving funds from the strategy and there are no funds being sent, this allowance consumption is unnecessary unless it is used to limit the number of retrievals, which is not specified. The check is necessary to enforce that only approved operators may trigger retrievals.

This appears to have been added as a mitigation to C4-1032. While the allowance check may be sufficient to enforce authorized delegated retrieval, the allowance spend may not be necessary.

Impact: If `msg.sender` is approved to have the required allowance (because it was granted for depositing into the strategy earlier), then it will be unnecessarily consumed. This scenario will either require the user to thereafter perform strategy retrieval themselves or cause loss of approved allowance for the delegated retriever.

Likelihood: Medium + Impact: Medium = Severity: Medium.

Recommendation: Consider removing the allowance consumption.

Tapioca: As we won't have cross chain strategies anymore..retrieve and send to strategy might not be used anymore....we'll remove it from TOFT.

Spearbit: Acknowledged that cross-chain strategies will be considered out-of-scope from this point onward in the review (was communicated as in-scope at review kick-off) and so will not review/report any more issues here.

5.3.42 Missing `unchecked` causes `FullMath.muldiv()` to revert instead of overflowing as expected during intermediate steps

Severity: Medium Risk

Context: twAML.sol#L2-L107, Code4rena#483, Uniswap V3, `tapioca-periph.FullMath.muldiv()`

Description: `FullMath.muldiv()` in `twAML.sol`, similar to original version from Remco and its adapted version Uniswap-V3, is expected to overflow during intermediate steps as commented in Uniswap V3:

> Handles "phantom overflow" i.e., allows multiplication and division where an intermediate value overflows 256 bits

However, given the Solidity version enforced via `pragma solidity ^0.8.18;` in `twAML.sol`, this does not happen because since Solidity 0.8.0, all arithmetic operations revert on overflow and underflow by default. This necessitates either an older compiler version (as done in Uniswap V3) or the use of `unchecked` over the entire `mulDiv()` function (as done in `tapioca-periph.FullMath.muldiv()`). While this was reported in the previous security review C4-483 and thereafter fixed in `tapioca-periph.FullMath.muldiv()`, this mitigation is missing here.

Impact: Calculations revert on expected "phantom" overflows in intermediate steps when they shouldn't.

Likelihood: Medium + Impact: Medium = Severity: Medium.

Recommendation: Add `unchecked` over the entire `mulDiv()` function (as done in `tapioca-periph.FullMath.muldiv()`).

Tapioca: Fixed in PR 120.

5.3.43 A stale `epochTAPValuation` will credit an incorrect TAP amount when users exercise their option

Severity: Medium Risk

Context: TapiocaOptionBroker.sol#L645, TapiocaOptionBroker.sol#L520

Description: When users exercise their oTAP options, `_processOTCDeal()` determines the user payment `otcAmountInUSD = tapAmount * epochTAPValuation;` using the value of `epochTAPValuation` which was updated in `newEpoch()`. But it is not guaranteed that `newEpoch()` has been called for this epoch and so `epochTAPValuation` may be a stale value from previous epochs.

Impact: A stale `epochTAPValuation` will credit an incorrect TAP amount when users exercise their option. Depending on the price movements, users may pay too much or too little. If there are big spikes in TAP prices across epochs when users exercise options, the impact may be quite severe if the stale value is used.

Likelihood: Medium + Impact: Medium = Severity: Medium.

Recommendation: Consider adding the below logic:

```
// Get epoch TAP valuation
bool success;
(success, epochTAPValuation) = tapOracle.get(tapOracleData);
if (!success) revert Failed();
```

in the beginning of `_processOTCDeal()` before `otcAmountInUSD = tapAmount * epochTAPValuation;`. Or, add logic to determine a stale value and revert.

Tapioca: Behaviour is correct, we take the epoch TAP price for option pricing, we don't compute the price at the time of exercise.

5.3.44 Incorrect threshold checks in `TapiocaOptionBroker.participate()` and `twTAP.participate()` prevent users locking positions for the maximum period

Severity: Medium Risk

Context: TapiocaOptionBroker.sol#L307-L308, oTAP Documentation, twTAP.sol#L301-L303, twTAP Documentation

Description: `TapiocaOptionBroker.participate()` computes the magnitude of the tOLP's lock duration against the current `pool.cumulative` and reverts when `if (magnitude >= pool.cumulative * 4)`. However, documentation mentions that:

> When a user locks their SGL receipt token and thus the underlying lending position, they can select any lock duration they wish, in units of epochs (weeks), with one epoch being the minimum escrow time, with a dynamic maximum which is always equal to four times the current AML.

`twTAP.participate()` similarly does the same, where its documentation mentions that:

> When locking TAP, twTAP receives TAP as an input, as well as a Time Weight to mint a certain amount of twTAP, according to the AML. The minimum lock time is one epoch (one week). The maximum lock time is dynamic, and is always four times the current AML.

So while `magnitude == pool.cumulative * 4` is the acceptable upper threshold, the enforced check reverts on this boundary condition.

Impact: This incorrect threshold check prevents users locking positions for the maximum period, which may be likely for users trying to maximize their oTAP DSO or twTAP incentives.

Likelihood: Medium + Impact: Medium = Severity: Medium.

Recommendation: Change TapiocaOptionBroker.sol#L307-L308 to:

```
  // Revert if the lock 4x the cumulative
- if (magnitude >= pool.cumulative * 4) revert TooLong();
+ if (magnitude > pool.cumulative * 4) revert TooLong();
```

Change twTAP.sol#L301-L303 to:

```
  // Revert if the lock 4x the cumulative
- if (magnitude >= pool.cumulative * 4) revert NotValid();
+ if (magnitude > pool.cumulative * 4) revert NotValid();
```

Tapioca: Fixed in PR 123.

5.3.45 Registering an already deployed BB market will lead to skipping of interest and fee accrual on it

Severity: Medium Risk

Context: Penrose.sol#L433-L445, Penrose.sol#L479-L491

Description: Penrose provides an `addBigBang()` function for the owner to register an already deployed BigBang market, as an alternative to `registerBigBang()` which both deploys and registers. However, `addBigBang()` misses adding this newly registered market `_contract` address to the `allBigBangMarkets` array. This `allBigBangMarkets` array is subsequently used in `reAccrueBigBangMarkets()` to call `accrue()` on all BigBang registered markets, which will therefore exclude all BB markets registered after deployment via `addBigBang()`.

Impact: Registering an already deployed BB market via `addBigBang()` will lead to skipping of interest and fee accrual on it and leading to loss of funds for protocol.

Likelihood: Medium + Impact: Medium = Severity: Medium.

Recommendation: Add `allBigBangMarkets.push(_contract);` to `addBigBang()`.

Tapioca: Fixed in PR 288.

5.3.46 Misplaced conversion causes incorrect liquidator rewards to be sent

Severity: Medium Risk

Context: Market.sol#L495-L502, Code4rena#1165

Description: The high-severity vulnerability from a previous security review Code4rena#1165 was mitigated by converting `borrowed` to include the accumulated fees via `borrowed = (borrowed * totalBorrow.elastic) / totalBorrow.base;` in `_getCallerReward()`.

However, this conversion was incorrectly placed after the comparisons of `borrowed` to `startTVLInAsset` and `maxTVLInAsset`. This effectively leads to a lower value of `borrowed` being used in the comparisons. In the border cases where:

1. `borrowed < startTVLInAsset`, zero rewards may be sent when it should actually be higher.
2. `borrowed >= maxTVLInAsset`, higher rewards are sent instead of `minLiquidatorReward`.

Impact: Liquidator will receive lower or higher rewards than intended by the protocol.

Likelihood: Medium + Impact: Medium = Severity: Medium.

Recommendation: Change to:

```
  function _getCallerReward(
      address user,
      uint256 _exchangeRate
  ) internal view returns (uint256) {
      (
          uint256 startTVLInAsset,
          uint256 maxTVLInAsset
      ) = _computeMaxAndMinLTVInAsset(
              userCollateralShare[user],
              _exchangeRate
          );

      uint256 borrowed = userBorrowPart[user];
      if (borrowed == 0) return 0;
      if (startTVLInAsset == 0) return 0;

+     borrowed = (borrowed * totalBorrow.elastic) / totalBorrow.base;

      if (borrowed < startTVLInAsset) return 0;
      if (borrowed >= maxTVLInAsset) return minLiquidatorReward;

-     borrowed = (borrowed * totalBorrow.elastic) / totalBorrow.base;

      uint256 rewardPercentage = ((borrowed - startTVLInAsset) *
          FEE_PRECISION) / (maxTVLInAsset - startTVLInAsset);

      int256 diff = int256(minLiquidatorReward) - int256(maxLiquidatorReward);
      int256 reward = (diff * int256(rewardPercentage)) /
          int256(FEE_PRECISION) +
          int256(maxLiquidatorReward);

      if (reward < int256(minLiquidatorReward)) {
          reward = int256(minLiquidatorReward);
      }

      return uint256(reward);
  }
```


Tapioca: Fixed in PR 283.

5.3.47 Stale `exchangeRate` can be forced across the protocol by repeatedly calling `updateExchangeRate()`

Severity: Medium Risk

Context: Market.sol#L366-L383, Market.sol#L60-L61

Description: `rateTimestamp` is specified as "latest timestamp when `exchangeRate` was updated" but is updated to `block.timestamp` whenever `updateExchangeRate()` is called, even if `exchangeRate` is not updated in the `else{}` block when `updated` is `false`.

Impact: Given this is a public function, one can keep calling it to update `rateTimestamp` and keep the rate valid beyond `rateValidDuration` as long as oracle has not updated the rate, i.e. stale `exchangeRate` can be force-used across the protocol within the oracle updation window by bypassing the `rateTimestamp + rateValidDuration >= block.timestamp` check.

Likelihood: Medium + Impact: Medium = Severity: Medium.

Recommendation: Move `rateTimestamp = block.timestamp;` within the `if` block:

```
  function updateExchangeRate() public returns (bool updated, uint256 rate) {
      (updated, rate) = oracle.get(oracleData);

      if (updated) {
          require(rate != 0, "Market: invalid rate");
          exchangeRate = rate;
+         rateTimestamp = block.timestamp;
          emit LogExchangeRate(rate);
      } else {
          require(
              rateTimestamp + rateValidDuration >= block.timestamp,
              "Market: rate too old"
          );
          // Return the old rate if fetching wasn't successful & rate isn't too old
          rate = exchangeRate;
      }

-      rateTimestamp = block.timestamp;
  }
```

Tapioca: Fixed in PR 282.

5.3.48 SGL liquidations may revert because of missing `yieldBox` approvals

Severity: Medium Risk

Context: SGLLiquidation.sol#L261-L283, BBLiquidation.sol#L233-L243

Description: The approvals to `yieldBox` are missing from the `if (callerShare > 0)` (unlike in `BBLiquidation`) because it is assumed that `feeShare > 0` and therefore we do not also need an approval in the `callerShare` logic.

However, in the boundary case that `feeShare == 0` when `extraShare == callerShare` (happens when `callerReward == FEE_PRECISION`) for any liquidation, the approvals in the `if (feeShare > 0)` block will not be triggered which will make `yieldBox.depositAsset()` revert for depositing of `callerShare`.

Impact: Liquidations will revert when they should not.

Likelihood: Low + Impact: High = Severity: Medium.

Recommendation: Add approvals to the `callerShare` logic as well:

```
  if (callerShare > 0) {
+     asset.approve(address(yieldBox), 0);
+     asset.approve(address(yieldBox), type(uint256).max);
      yieldBox.depositAsset(
          assetId,
          address(this),
          msg.sender,
          0,
          callerShare
      );
  }
```

Tapioca: Fixed in PR 280.

5.3.49 Unsafe downcasting may lead to unexpected overflows

Severity: Medium Risk

Context: BBCommon.sol#L71, BBCommon.sol#L95, BBCommon.sol#L69, BBCommon.sol#L83, , Market.sol#L507, BBLiquidation.sol#L217-L218, SGLCommon.sol#L115, SGLCommon.sol#L258, twTAP.sol#L488, TapiocaOptionBroker.sol#L244-L246, TapiocaOptionBroker.sol#L471-L473

Description: Several expressions across the protocol perform unsafe downcasting, for example, from `uint256` to `uint128`, `uint64` or `int256`, some of which are referenced above for context.

Impact: Unless it is provably safe, this may lead to unexpected silent overflows leading to use of truncated incorrect values affecting the related logic.

Likelihood: Low + Impact: High = Severity: Medium

Recommendation: Evaluate all instances of unsafe downcasting (above references are not exhaustive) and consider using OpenZeppelin's SafeCast.

Tapioca: Fixed in PR 279.

5.3.50 `USDOFlashloanHelper.flashloan` is breaking EIP3156 by not consuming all `allowance`

Severity: Medium Risk

Context: USDOFlashloanHelper.sol#L128-L160

Description: The call to flashloan will handle burning as follows (see USDOFlashloanHelper.sol#L128-L160):

```
// Stack to deep
// usdo.burn(address(receiver), amount)
assembly {
    // Free memory pointer
    let freeMemPointer := mload(0x40)

    // keccak256("burn(address,uint256)")
    mstore(freeMemPointer, shl(224, 0x9dc29fac))

    mstore(add(freeMemPointer, 4), receiver)
    mstore(add(freeMemPointer, 36), amount)

    // Execute the call
    let success := call(
        gas(), // Send all gas
        token, // The address of the usdo contract
        0, // No ether is sent
        freeMemPointer, // Input pointer
        68, // Input length (4 bytes for method ID + 32 bytes for address + 32 bytes for uint256)
        0,
        0
    )

    // Check for failure and revert
    if iszero(success) {
        revert(0, 0)
    }

    // Adjust the free memory pointer
    mstore(0x40, add(freeMemPointer, 68))
}

usdo.transferFrom(address(receiver), address(usdo), fee); /// @audit This is spending the allowance
```

Which doesn't consume the allowance of the caller, breaking this MUST from EIP 3156.

Recommendation: Acknowledge the non-compliance or consume the allowance by first transferring all tokens to self, and then burning them.

Tapioca: Fixed by transferring `amount` to self and then burning it.

Spearbit: Verified.

5.3.51 Double allowance spend leads to loss of funds for spenders across all USDO/TOFT YB, lending, option, strategy and leverage markets

Severity: Medium Risk

Context: USDOMarketModule.sol#L118-L128, USDOOptionsModule.sol#L56-L73, USDOLeverageModule.sol#L38-L57, OFTV2.sol#L34, BaseTOFTGenericModule.sol#L54-L67, BaseTOFTMarketModule.sol#L127-L136, BaseTOFTOptionsModule.sol#L64-L86, BaseTOFTStrategyModule.sol#L61-L68, BaseTOFTLeverageModule.sol#L51-L76

Description: USDO/TOFT YB, lending, option, strategy and leverage markets check if `_from != msg.sender` and make a call to `_spendAllowance()` for deducting the allowance amount approved to spender. However, they subsequently also make a call to `OFTV2._debitFrom()` which also calls `_spendAllowance()` with the same parameters.

Impact: Double allowance spend leads to loss of funds for spenders across all USDO/TOFT YB, lending, option, strategy and leverage markets in the case of non-infinite allowance.

Likelihood: Medium + Impact: Medium = Severity: Medium

Recommendation: Consider removing the explicit call to `_spendAllowance()` and rely on `_debitFrom()` for deducting the allowance amount approved to spender.

Tapioca: Fixed in PRs 272 and 116.

5.3.52 Using `approve` could revert for certain tokens

Severity: Medium Risk

Context: Penrose.sol#L524-L525

Description: This is a very common finding, worth fixing as the markets are generalized, and there's no downside in doing so. `approve` will expect a return value, tokens (such as USDT) that don't return it will revert when performing the call (see Penrose.sol#L524-L525):

```
IERC20(_asset).approve(address(twTap), 0);
IERC20(_asset).approve(address(twTap), feeAmount);
```

This applies to a vast number of instances in the codebase. A quick find-replace should help solve this.

Recommendation: Use `safeApprove`.

Tapioca: Replaced `approve` with `safeApprove` in PR 268.

5.3.53 `BBLendingCommon` is not minting fees which breaks an implicit invariant of CDP Systems

Severity: Medium Risk

Context: BBLendingCommon.sol#L59-L60

Description: Minting USDO via `BigBang` will trigger a fee (see BBLendingCommon.sol#L59-L60):

```
(totalBorrow, part) = totalBorrow.add(amount + feeAmount, true); /// @audit TODO BIG: Total vs Part
```

This fee is debited to the user, but it is not minted anywhere (see BBLendingCommon.sol#L74-L75):

```
IUSDOBase(address(asset)).mint(address(this), amount); /// @audit Broken invariant, total deb >> total
↪  supply
```

This will break an important invariant of CDP Systems, that the `totalDebt` is equal to the `totalSupply`. One key reason to maintain said invariant is that in the case of a system migration, perhaps to V2, some people will be unable to close their positions as they will have to pay some debt that doesn't exist.

In contrast, see how Liquity handles the fee by distributing it to stakers (see BorrowerOperations.sol#L363-L374):

```
function _triggerBorrowingFee(ITroveManager _troveManager, ILUSDToken _lusdToken, uint _LUSDAmount,
↪  uint _maxFeePercentage) internal returns (uint) {
    _troveManager.decayBaseRateFromBorrowing(); // decay the baseRate state variable
    uint LUSDFee = _troveManager.getBorrowingFee(_LUSDAmount);

    _requireUserAcceptsFee(LUSDFee, _LUSDAmount, _maxFeePercentage);

    // Send fee to LQTY staking contract
    lqtyStaking.increaseF_LUSD(LUSDFee);
    _lusdToken.mint(lqtyStakingAddress, LUSDFee);

    return LUSDFee;
}
```


For commentary on the invariant, please also check C4-1276.

Tapioca-DAO: Fixed in PRs 150 and 285.

5.3.54 `liquidate` relying on `MarketLiquidatorReceiver` will fail when the oracle is not `updated`

Severity: Medium Risk

Context: MarketLiquidatorReceiver.sol#L138C1-L146C71

Description: Liquidations are meant to tolerate faults from the oracle via the following logic (see BBLiquidation.sol#L99-L106):

```
// Oracle can fail but we still need to allow liquidations
(bool updated, uint256 _exchangeRate) = oracle.get(oracleData);
if (updated && _exchangeRate > 0) {
    exchangeRate = _exchangeRate; //update cached rate
} else {
    _exchangeRate = exchangeRate; //use stored rate
}
if (_exchangeRate == 0) revert ExchangeRateNotValid();
```

However, in the same logic path, `MarketLiquidationReceiver.onCollateralReceiver` will be called, which in turn, will verify the `_minOut` via (see MarketLiquidatorReceiver.sol#L138C1-L146C71):

```
function _getMinAmount(
    address _tokenIn,
    uint256 tokenInAmount,
    uint256 _slippage
) private returns (uint256 minTokenOutAmount) {
    IOracle oracle = IOracle(oracles[_tokenIn].target);
    (bool updated, uint256 rate) = oracle.get(oracles[_tokenIn].data);
    require(updated, "MarketLiquidatorReceiver: oracle called failed");
    require(rate > 0, "MarketLiquidatorReceiver: rate not valid");
```

In the case of a non update oracle, this call will revert, meaning that liquidations can only be performed when the oracle is `updated`.

Recommendation: Since the Path is trusted, and the caller is enforced, you could just pass the price from the BB/SGLiquidation module. I would recommend removing this swap path as it will be inefficient and a ton of work to maintain, however the findings highlights a failure case that should be solved in either case.

Tapioca: I prefer not to pass it from outside to keep the code and logic of the liquidator receiver separate of the market code. Also some users might use the same liquidator receiver for multiple tokens (as the current `MarketLiquidatorReceiver.sol` contract). However, I used the same approach we used in market's liquidation method by keeping a cached rate, in PR 267.

5.3.55 `setBigBangEthMarketDebtRate` will retroactively apply the new rate to the ETH market - Must accrue first

Severity: Medium Risk

Context: Penrose.sol#L276-L279

Description: The BigBang ETH market rate can be changed by the owner via (see Penrose.sol#L276-L279):


```
function setBigBangEthMarketDebtRate(uint256 _rate) external onlyOwner {
    bigBangEthDebtRate = _rate; /// @audit Must accrue First
    emit BigBangEthMarketDebtRate(_rate);
}
```

Because `accrue` simply checks the rate and applies it from the last time of accrue, if you don't call `accrue` before updating the rate, you'll end up retroactively applying on the next call to `accrue`, this can cause unintended consequences such as a overly high or incorrectly low borrow fee being applied retroactively.

Recommendation: Add a call to `ethMarket.accrue()` before updating the `ethDebtRate`.

Tapioca: Fixed it in PR 261.

Spearbit: Tapioca has fixed it by calling `accrue` on the `bigBangEthMarket` before changing the rate.

5.3.56 New singularity configuration can be applied backwards

Severity: Medium Risk

Context: Singularity.sol#L511-L523

Description: No update is performed before setting new configuration values, which are used in the interest rate logic, so they will be applied backwards for the period from the last `_accrue()` call to the next one, while `setSingularityConfig()` happened somewhere in-between. This period's length is not controlled on-chain.

Configuration parameters (e.g. min and max `InterestPerSecond` and `TargetUtilization`, `interestElasticity`) are used in interest rate calculation:

- SGLCommon.sol#L37-L149

```
function _getInterestRate()
    internal
    view
    returns (
        // ...
    )
{
    // ...

    // Update interest rate
    if (utilization < minimumTargetUtilization) {
        uint256 underFactor = ((minimumTargetUtilization - utilization) *
            FACTOR_PRECISION) / minimumTargetUtilization;
        uint256 scale = interestElasticity +
            (underFactor * underFactor * elapsedTime);
        _accrueInfo.interestPerSecond = uint64(
            (uint256(_accrueInfo.interestPerSecond) * interestElasticity) /
                scale
        );
        if (_accrueInfo.interestPerSecond < minimumInterestPerSecond) {
            _accrueInfo.interestPerSecond = minimumInterestPerSecond; // 0.25% APR minimum
        }
    } else if (utilization > maximumTargetUtilization) {
        uint256 overFactor = ((utilization - maximumTargetUtilization) *
            FACTOR_PRECISION) / fullUtilizationMinusMax;
        uint256 scale = interestElasticity +
            (overFactor * overFactor * elapsedTime);
        uint256 newInterestPerSecond = (uint256(
            _accrueInfo.interestPerSecond
        ) * scale) / interestElasticity;
        if (newInterestPerSecond > maximumInterestPerSecond) {
            newInterestPerSecond = maximumInterestPerSecond; // 1000% APR maximum
        }
        _accrueInfo.interestPerSecond = uint64(newInterestPerSecond);
    }
}
```

Impact: backwards applicability means using a misstated set of the parameters for that period, during which the system will follow an incorrect logic.

Recommendation: Consider calling `_accrue()` before the update, e.g. Singularity.sol#L511-L523:

```
  function setSingularityConfig(
      // ...
  ) external onlyOwner {
+     _accrue();
      if (_borrowOpeningFee > FEE_PRECISION) revert NotValid();
      emit LogBorrowingFee(borrowOpeningFee, _borrowOpeningFee);
      borrowOpeningFee = _borrowOpeningFee;
```

Tapioca: Fixed in PR 256.

Spearbit: Fix looks ok, `_accrue()` is now called before configuration update.

5.3.57 Boundary opening fees are misstated

Severity: Medium Risk

Context: BBLendingCommon.sol#L89-L103

Description: Boundary fees aren't applied to the amount in question and aren't converted from fee precision to token precision. Due to this whenever the boundaries are hit the effective fees being reduced to be fixed dust amounts.

Since `_computeVariableOpeningFee()` returned fee amount is deemed to be in token units:

- BBBorrow.sol#L39-L45:

```
uint256 feeAmount = _computeVariableOpeningFee(amount);
uint256 allowanceShare = _computeAllowanceAmountInAsset(
    from,
    exchangeRate,
    // see the line below
    amount + feeAmount,
    asset.safeDecimals()
);
```

- BBLendingCommon.sol#L51-L59

```
function _borrow(
    address from,
    address to,
    uint256 amount,
    uint256 feeAmount
) internal returns (uint256 part, uint256 share) {
    openingFees[to] += feeAmount;
// see the line below
    (totalBorrow, part) = totalBorrow.add(amount + feeAmount, true);
```


But plain fee levels are returned on boundaries:

- BBLendingCommon.sol#L89-L101

```
if (_exchangeRate >= minMintFeeStart) return minMintFee;
if (_exchangeRate <= maxMintFeeStart) return maxMintFee;

// ...

if (fee > maxMintFee) return maxMintFee;
if (fee < minMintFee) return minMintFee;

//...
```

- BBStorage.sol#L49-L50

```
uint256 public minMintFee = 0;
uint256 public maxMintFee = 1000;
```

Impact: the fees become effectively zero when USDO/USDC exchange rate reaches boundary `minMintFeeStart`, `maxMintFeeStart` levels.

Recommendation: Consider translating the fee levels to correspond to token dp and amount on boundaries, e.g. BBLendingCommon.sol#L89-L103:

```
- if (_exchangeRate >= minMintFeeStart) return minMintFee;
- if (_exchangeRate <= maxMintFeeStart) return maxMintFee;
+ if (_exchangeRate >= minMintFeeStart) return (amount * minMintFee) / FEE_PRECISION;
+ if (_exchangeRate <= maxMintFeeStart) return (amount * maxMintFee) / FEE_PRECISION;

  uint256 fee = maxMintFee -
      (((_exchangeRate - maxMintFeeStart) * (maxMintFee - minMintFee)) /
          (minMintFeeStart - maxMintFeeStart));

- if (fee > maxMintFee) return maxMintFee;
- if (fee < minMintFee) return minMintFee;
+ if (fee > maxMintFee) return (amount * maxMintFee) / FEE_PRECISION;
+ if (fee < minMintFee) return (amount * minMintFee) / FEE_PRECISION;

  if (fee > 0) {
      return (amount * fee) / FEE_PRECISION;
  }
```

Tapioca: Fixed it in PR 257.

Spearbit: Fix looks ok.

5.4 Low Risk

5.4.1 `twTap.participate{gas: 310_000}(to, amount, duration)` is griefable by `to`

Severity: Low Risk

Context: BaseTapOFT.sol#L163-L165

Description: `_lockTwTapPosition` calls BaseTapOFT.sol#L163-L165:

```
try /// @audit Exact Gas is unacceptable here due to `_safeMint`
    twTap.participate{gas: 310_000}(to, amount, duration) // Should consume 300_848 gas
{} catch Error(string memory _reason) {
```

Which will trigger a `safeMint` in twTAP.sol#L348-L349:

```
_safeMint(_participant, tokenId); /// @audit `safeMint` in the middle = Reenter + Burn all gas
```

This could allow the `_participant` which is `to` to spend more gas than intended, causing the call to fail.

Recommendation: Consider whether failure in this scenario is acceptable or whether the call should forward some extra gas for the `to`.

Tapioca: Acknowledged. There's no benefit for the user in causing the call to fail.

Spearbit: Acknowledged.

5.4.2 Registering a user multiple times will overwrite airdrop amounts

Severity: Low Risk

Context: AirdropBroker.sol#L358-L374

Description: `registerUserForPhase()` accepts arrays of airdrop users and amounts for phases one and four. However, it assumes that the `_users` array does not accidentally have any duplicate addresses.

Impact: Registering an user address that accidentally appears multiple times in `_users` will overwrite airdrop amounts for that user to the last overwritten value.

Likelihood: Low + Impact: Low (If detected and registration is repeated again correctly) = Severity: Low.

Recommendation: Consider checking for duplicate user addresses as a defensive measure.

Tapioca: Acknowledged.

Spearbit: Acknowledged.

5.4.3 `computeTimeFromAmount()` returning the time offset may be unexpected

Severity: Low Risk

Context: Vesting.sol#L108-L120, Vesting.sol#L246, Vesting.sol#L226-L231

Description: `computeTimeFromAmount()` is documented as returning "*Compute the time needed to unlock an amount of tokens, given a total amount.*" However, in the computation of `_computeTimeFromAmount()`, `_start - (_start - ((_amount * _duration) / _totalAmount))`, `_start` effectively is ignored to return only the time offset of `(_amount * _duration) / _totalAmount)`. While this is accurate for the computation of `__initialUnlockTimeOffset`, it may return an unexpected value to anyone calling the getter `computeTimeFromAmount()`.

Impact: `computeTimeFromAmount()` returning the time offset may be unexpected and lead to user confusion while determining vesting unlock times.

Likelihood: High + Impact: Very Low = Severity: Low.

Recommendation: Consider removing `_start` parameter or reevaluate what this function is actually expected to return.

Tapioca: Addressed in commit 220608e7.

5.4.4 `AirdropBroker.newEpoch` may allow reentrancy via the oracle read

Severity: Low Risk

Context: AirdropBroker.sol#L320-L324

Description: From tapioca-periphery, we know that `oracle.get` is not a view function:

```
function get(
    bytes calldata
)
    external
    virtual
    override
    nonReentrant
    returns (bool success, uint256 rate)
{
```

In the case that the oracle were to allow an attacker to regain control. The order of operations of `AirdropBroker.newEpoch` would allow to act as if the new epoch had started, while retaining the old price:

```
// Get epoch TAP valuation
(bool success, uint256 _epochTAPValuation) = tapOracle.get( /// @audit Could this be done on purpose to
↪  mess up changes?
    tapOracleData /// @audit Also note that the oracle could update after, and could be non-view so
↪  reentrancy here may be used to use old valuation
);
```

Recommendation: Consider a different order of operations as to prevent changes mid execution. It may be exceptionally best to have the oracle being called as the first thing, meaning no other storage changes would have happened and the potential reentrancy would happen on the older "normal" values.

5.4.5 Precision loss / lack of decimal adjustment in applying `TAPValuation`

Severity: Low Risk

Context: AirdropBroker.sol#L321-L326

Description: The following logic leads to storing a `epochTAPValuation` that will be multiplied by an amount to lead to a price:

```
// Get epoch TAP valuation
(bool success, uint256 _epochTAPValuation) = tapOracle.get( /// @audit Could this be done on purpose to
↪  mess up changes?
    tapOracleData /// @audit Also note that the oracle could update after, and could be non-view so
↪  reentrancy here may be used to use old valuation
);
if (!success) revert Failed();
epochTAPValuation = uint128(_epochTAPValuation);
```

This means that `epochTAPValuation` will be the price of a wei of Tapioca token, which will result in substantial precision loss.

Recommendation: Defining a specific "Oracle Precision" will ensure that you can store a more precise value:

```
amount * epochTAPValuation / ORACLE_PRECISION
```

5.4.6 Protocol parameters being different from final deployment configuration may lead to unexpected behavior

Severity: Low Risk

Context: BigBang.sol#L183

Description: Protocol parameters should be tested and reviewed with the final deployment configuration. If not, this may lead to unexpected behavior after deployment.

For example, `BigBang.protocolFee` is set to 0 for now but expected to be 10% when deployed, per the code comment.

Impact: Protocol parameters being different from final deployment configuration not only affects reviewability but also may lead to unexpected behavior and require owners to pause the protocol and/or call appropriate setters to reset their values. Parameters without setters may require contract redeployment.

Likelihood: Low + Impact: Medium (Assuming no major deviations leading to loss/lock of protocol funds) = Severity: Low.

Recommendation: Review all protocol parameters to ensure that they are initialized with the same values meant for deployment before testing.

Tapioca: Acknowledged.

Spearbit: Acknowledged.

5.4.7 Missing checks in setters may lead to unexpected behavior

Severity: Low Risk

Context: BigBang.sol#L442-L443, BigBang.sol#L455-L456

Description: Setters of critical protocol parameters should enforce sanity and threshold checks to ensure that new values are within expected/reasonable bounds and satisfy any implicit invariants against other protocol parameters. Allowing accidental setting of such parameters to absurd values may lead to unexpected behavior.

For example, `setMinAndMaxMintRange()` is missing a sanity check for `minMintFeeStart >= maxMintFeeStart` and `setMinAndMaxMintFee()` is missing a sanity check for `maxMintFee >= minMintFee`.

Likelihood: Low + Impact: Medium = Severity: Low.

Recommendation: Consider adding appropriate sanity checks to all setters of critical protocol parameters.

Tapioca: Addressed in PR 325.

5.4.8 `USDOFlashloanHelper.flashloan` deviating from ERC3156 MUST on `maxFlashLoan` may affect receiver integrations

Severity: Low Risk

Context: USDOFlashloanHelper.sol#L66-L73, ERC3165 Lender Specification

Description: ERC-3165 Lender Specification says that: "*The `maxFlashLoan` function MUST return the maximum loan possible for token. If a token is not currently supported `maxFlashLoan` MUST return 0, instead of reverting*".

`USDOFlashloanHelper.sol` enables flash loans only for USDO. However, `maxFlashLoan(address)` ignores the token passed and returns a value assuming that the requested flash loan token is USDO, which may not be the case if the integrator is not aware of this aspect.

Impact: `USDOFlashloanHelper.flashloan` deviating from EIP3156 MUST on `maxFlashLoan` will affect receiver integrations that attempt to borrow a non-USDO token for some reason.

Likelihood: Low + Impact: Low = Severity: Low.

Recommendation: Consider changing the implementation to `maxFlashLoan(address token)`, check for `token != address(usdo)` (as done in `flashFee()`) to return 0 for non-USDO tokens.

Tapioca: Addressed in PR 322.

5.4.9 `sendForLeverage()` logging incorrect sender and receiver addresses may lead to mismatched accounting

Severity: Low Risk

Context: USDOLeverageModule.sol#L88, BaseTOFTLeverageModule.sol#L107

Description: USDO and TOFT `sendForLeverage()` emit a `SendToChain` event to log the amount sent for leverage along with the sender and receiver addresses on the source and destination chains respectively.

In the context of the ussue "Incorrect refund address causes loss of cross-chain `sendForLeverage()` amount to users", it was specified that `msg.sender` will hereafter be treated only as an approved address/operator to therefore debit and credit the leveraged amount from `leverageFor` on the source and destination chains respectively. This means that `SendToChain` should use `leverageFor` for both its sender and receiver addresses. However, `msg.sender` is incorrectly being used now.

Impact: If these events are used for cross-layer bookkeeping, they will lead to mismatched accounting.

Likelihood: High + Impact: Very Low (Depends on how the offchain tooling/accounting uses this event) = Severity: Low.

Recommendation: Use `leverageFor` for both sender and receiver addresses in `SendToChain` event.

Tapioca: Addressed in PR 322.

5.4.10 `USDOGenericModule.triggerSendFrom()` using destination chain's `zroPaymentAddress` on source chain may revert

Severity: Low Risk

Context: USDOGenericModule.sol#L107, BaseTOFTGenericModule.sol#L204-L237

Description: LayerZero potentially supports paying for gas in its ZRO token at some point based on the current V1 implementation, e.g. code comment: "*zroPaymentAddress set to address(0x0) if not paying in ZRO (LayerZero Token)*". Cross-chain transactions therefore take a `_zroPaymentAddress` parameter in `_lzSend()` which would be the address funding the ZRO tokens for gas.

`USDOGenericModule.triggerSendFrom()` triggers the destination chain to send back the specified USDO amount to the source chain and provides a `sendFromData` parameter to be used with `sendFrom()` on the destination chain. However, the `sendFromData.zroPaymentAddress` meant to be used on the destination chain is used in the `_lzSend()` call on the source chain which may not be what the user intended and may revert if it does not have the required ZRO tokens also on the source chain.

`BaseTOFTGenericModule.triggerSendFrom()`, which implements a similar flow for generic TOFTs, requires the user to therefore specify an additional `zroPaymentAddress` parameter to be used on the source chain which is separate from `sendFromData.zroPaymentAddress` meant for the destination chain.

Impact: `USDOGenericModule.triggerSendFrom()` using destination chain's `sendFromData.zroPaymentAddress` on source chain may revert if it does not have the required ZRO tokens.

Likelihood: Low (Requires using ZRO for gas) + Impact: Medium = Severity: Low.

Recommendation: Add an additional `zroPaymentAddress` parameter to `USDOGenericModule.triggerSendFrom()` to be used with `_lzSend()`.

Tapioca: Acknowledged. The fee payment structure changes in V2.

Spearbit: Acknowledged.

5.4.11 A high maximum swap slippage may cause an unexpected loss during `sendForLeverage()` on the destination chain

Severity: Low Risk

Context: BaseUSDOStorage.sol#L53, USDOLeverageModule.sol#L91-L98, USDOLeverageModule.sol#L28-L41, BaseTOFTStorage.sol#L93-L95

Description: Unlike TOFTs where `SWAP_MAX_SLIPPAGE` initialized to 5% is configurable later via `setMaxSlippage()`, USDO has `SWAP_MAX_SLIPPAGE` as a constant set to 5%. This is used in `_assureMaxSlippage()` to ensure user-acceptable `minAmount` of `swapData.tokenOut` during the swap. Even for cross-chain swaps, 5% may be an unreasonably high slippage tolerance.

Impact: While this is incorrectly applying slippage check on the source chain, as raised in the issue "`USDO.sendForLeverage` will not work as it's incorrectly applying slippage checks", a 5% maximum swap slippage constant may cause an unexpected loss due to high slippage during `sendForLeverage()` if this is enforced on the destination chain.

Likelihood: Low + Impact: Low = Severity: Low.

Recommendation: Consider changing the default to a lower value and adding a setter to modify it later as done for TOFTs.

Tapioca: `_assureMaxSlippage` doesn't exist anymore. It was removed in a previous PR.

5.4.12 USDO deployer retaining minter/burner roles may be risky

Severity: Low Risk

Context: BaseUSDOStorage.sol#L74-L75, BaseUSDO.sol#L151, USDO.sol#L76-L87

Description: The constructor of `BaseUSDOStorage` grants minter and burner roles to the deployer of USDO after which `BaseUSDO`'s constructor transfers ownership to the provided `_owner` argument.

The original deployer retaining USDO minter/burner roles may be risky. Unless these are revoked immediately using `setMinterStatus()` and `setBurnerStatus()`, stale permissions may be exploited later if the deployer key is somehow compromised. For example, Ankr was similarly exploited because of a compromised deployer key allowing the attacker to mint billions of tokens.

Keys managing owner roles are expected to be reasonable multisigs backed by hardware wallets, whereas deployer keys may not necessarily have the same level of protection post-deployment.

Impact: USDO minting and burning may be taken over by a compromised deployer key if its minting/burning roles are not revoked.

Likelihood: Very Low (Requires compromised deployer key and retaining of roles) + Impact: High (arbitrary minting/burning) = Severity: Low.

Recommendation: Consider passing `_owner` to `BaseUSDOStorage` constructor and assigning minter/burner roles to that instead.

Tapioca: Addressed in PR 322.

5.4.13 Whenever main ETH market is changed in `Penrose`, the linked markets will have incorrect interest rate calculations for the last period as they aren't accrued first

Severity: Low Risk

Context: Penrose.sol#L281-L286

Description: On any configuration change from one non-zero main ETH market implementation to another (which is presumably rare, but possible in the current logic), it's needed to accrue all linked non-ETH markets first similarly to `reAccrueBigBangMarkets()`, otherwise new ETH market parameters will be incorrectly used in the linked markets accrual logic for some periods before `setBigBangEthMarket()` was run, i.e. back propagated:

```
/// @notice sets the main BigBang market
/// @dev needed for the variable debt computation
function setBigBangEthMarket(address _market) external onlyOwner {
    // see the line below
    bigBangEthMarket = _market;
    emit BigBangEthMarketSet(_market);
}
```

As linked markets interest rate logic depends on the ETH market state (see BBCommon.sol#L24-L32):

```
function getDebtRate() public view returns (uint256) {
    if (isMainMarket) return penrose.bigBangEthDebtRate(); // default 0.5%
    if (totalBorrow.elastic == 0) return minDebtRate;
    // see the line below
    uint256 _ethMarketTotalDebt = IBigBang(penrose.bigBangEthMarket())
        .getTotalDebt();
    uint256 _currentDebt = totalBorrow.elastic;
    // see the line below
    uint256 _maxDebtPoint = (_ethMarketTotalDebt *
        debtRateAgainstEthMarket) / 1e18;
```

Impact: linked markets will apply the updated interest rate (based on the new ETH market) to the pre-update period, so they will accrue at the incorrect rate between the previous `_accrue()` calling operation there and `setBigBangEthMarket()`.

Likelihood: very Low (per rarity of the operation by itself) + Impact: High (incorrect accruals across all the linked markets) = Severity: Low.

Recommendation: Consider updating the linked markets whenever it is a change from a non-zero implementation, e.g. (from `reAccrueBigBangMarkets()` in Penrose.sol#L281-L286):

```
  /// @notice sets the main BigBang market
  /// @dev needed for the variable debt computation
  function setBigBangEthMarket(address _market) external onlyOwner {
+     if (bigBangEthMarket != address(0)) {
+          uint256 len = allBigBangMarkets.length;
+          address[] memory markets = allBigBangMarkets;
+          for (uint256 i = 0; i < len; i++) {
+             address market = markets[i];
+             if (market != bigBangEthMarket && isMarketRegistered[market]) {
+                 IBigBang(market).accrue();
+             }
+         }
+     }
      bigBangEthMarket = _market;
      emit BigBangEthMarketSet(_market);
  }
```

It is also advised to move all this additional code to a new internal function and call it from here and from `reAccrueBigBangMarkets()`.

Tapioca: Fixed in PR 322.

Spearbit: Fix looks ok.

5.4.14 Yearly interest is ignoring leap years, overcharging slightly

Severity: Low Risk

Context: BBCommon.sol#L83-L84

Description: `BBCommon` uses `31536000` for seconds in a year, which maps out to 365 days

```
_accrueInfo.debtRate = uint64(annumDebtRate / 31536000); //per second
```

Instead 31557600, which is 365.25, is accounting for leap years

Recommendation: Consider changing 31536000 to 31557600 to account for leap years

Tapioca: Fixed in PR 322.

5.4.15 `TapiocaWrapper.executeCalls` can have multiple `success` and `failure` but only returns the last

Severity: Low Risk

Context: TapiocaWrapper.sol#L111-L112

Description: `executeCalls` has the following signature (see TapiocaWrapper.sol#L105C14-L112):

```
 function executeCalls(
    ExecutionCall[] calldata _call
)
    external
    payable
    onlyOwner
    returns (bool success, bytes[] memory results)
{
```

And will update `success` with the last result (see TapiocaWrapper.sol#L119-L121):

```
(success, results[i]) = payable(_call[i].toft).call{
    value: _call[i].value
}(_call[i].bytecode);
```

This will make it impossible to integrating / calling smart contracts to determine which call failed.

Recommendation: Create and return an array of `success` value for each call

Tapioca: Acknowledged. Did it on `lzv2-migration` branch.

Spearbit: Acknowledged.

5.4.16 `BaseTOFT._wrap` will not work for tokens that charge a `feeOnTransfer`

Severity: Low Risk

Context: BaseTOFT.sol#L557-L558

Description: `BaseTOFT._wrap` credits `_amount` to the end user, but a token with Fee On Transfer may result in a lower amount received by the contract (see BaseTOFT.sol#L557-L558):

```
IERC20(erc20).safeTransferFrom(_fromAddress, address(vault), _amount);
_mint(_toAddress, _amount);
```

This could cause losses to the last depositor and potential unintended reverts.

Recommendation: Acknowledge the issue and document it for integrators or consider applying a delta balance check to determine the actual amount received. Also, consider that supporting fee on transfer tokens xChain will present further challenges that are most likely not worth the additional logic and smart contract risk.

Tapioca: Acknowledged. We'll probably not use such types of tokens, but I will chat with everyone to make sure.

Spearbit: Acknowledged.

5.4.17 Minimum range configurations for `oTAP` and `twTAP` are higher than the 0% specified

Severity: Low Risk

Context: TapiocaOptionBroker.sol#L93, oTAP Documentation, twTAP.sol#L102, twTAP Documentation

Description: The range configuration for oTAP discount is specified as 0% to 50%. Similarly, the range for twTAP is specified as 0% to 100%. However, the oTAP implementation enforces a range of 5% to 50%, while that for twTAP enforces a range of 10% to 100%. It is unclear if this is intended or not, but the implementation deviates from the documented specification for the lower bound.

Impact: Minimum range configurations for oTAP and twTAP are higher than the 0% specified potentially causing a loss to protocol.

Likelihood: Low (Assuming documentation error) + Impact: Medium = Severity: Low.

Recommendation: Evaluate if this is intended or not to fix the implementation or documentation accordingly.

Tapioca: Addressed in PR 152.

5.4.18 Allowing `setPhase2MerkleRoots()` to be called after start of second phase may affect specified `aoTAP` distribution

Severity: Low Risk

Context: AirdropBroker.sol#L352-L356, AirdropBroker.sol#L241-L242, AirdropBroker.sol#L449-L472

Description: The second phase of `aoTAP` distribution uses merkle proofs to determine participant inclusion in the four different roles of Tapioca Guild. The owner is expected to set `phase2MerkleRoots` for the four roles using `setPhase2MerkleRoots()` before the start of second phase. However, the implementation does not check the current phase/epoch and allows owner to change `phase2MerkleRoots` again after the start of second phase.

Impact: Owner may accidentally affect the specified distribution by changing `phase2MerkleRoots` after the start of second phase.

Likelihood: Low (requires owner to accidentally call `setPhase2MerkleRoots()` again) + Impact: Medium = Severity: Low.

Recommendation: Add a check in `setPhase2MerkleRoots()` to ensure that `epoch <= 1`. Similar safeguards can be added to `registerUserForPhase()`.

Tapioca: Addressed in PR 150. Confirming for a part of registerUserForPhase(), but in this case only phase 1 needs to be checked, as we want to be able to set that on subsequent epochs.

5.4.19 Allowing anyone to donate reward tokens to `twTAP` may cause unexpected behavior

Severity: Low Risk

Context: twTAP.sol#L473-L492, Penrose.sol#L526

Description: Penrose calls `twTAP.distributeReward()` in `Penrose.withdrawAllMarketFees()` to distribute market fees to `twTAP` for its participants to thereafter claim them as rewards. However, `twTAP.distributeReward()` does not enforce that only Penrose is allowed to call it, which allows anyone to donate reward tokens.

Impact: While any donations will cause an accounting mismatch between the reward token amount in `twTAP` versus those logged by Penrose `LogTwTapFeesDeposit(feeShares, feeAmount)`, it is not clear that this will not cause other unexpected behavior.

Likelihood: Low (Requires donating reward tokens) + Impact: Low (Assuming no other unexpected behavior other than accounting mismatch) = Severity: Low.

Recommendation: Consider allowing only Penrose to call `twTAP.distributeReward()`. Nit: Consider logging `_assetId` instead of `feeShares` in `LogTwTapFeesDeposit(feeShares, feeAmount)` to indicate the specific reward token.

Tapioca: Acknowledged. We want to be able to send rewards in the future from multiple source. As an issue I think the severity is super low as you mentioned, it requires donating tokens, and there are no other action taken within the function, except increasing the awarded tokens.

Spearbit: Acknowledged.

5.4.20 Anyone being allowed to force-claim rewards for any `twTAP` may lead to unexpected behavior

Severity: Low Risk

Context: twTAP.sol#L394, twTAP.sol#L518-L531

Description: As commented, `_requireClaimPermission()` is supposed to mirror the access checks of `_isApprovedOrOwner()` but additionally give permission if `_to` is the owner. This allows anyone besides owners, operators & token approved addresses to force-claim rewards for any `twTAP` with the reward tokens being transferred to the owner.

Impact: This may cause unexpected behavior with tracking and accounting if the `twTAP` owner has explicitly approved other entities to manage claiming and receiving rewards on their behalf.

Likelihood: High (anyone can trigger `claimRewards()`) + Impact: Very Low (unexpected behavior with tracking and accounting) = Severity: Low.

Recommendation: Consider removing the modification in `_requireClaimPermission()` to allow if `_to` is the owner to make it consistent only with `_isApprovedOrOwner()`.

Tapioca: Addressed in PR 148.

5.4.21 Missing `_storeFailedMessage()` in `catch Error` block of `BaseTapOFT._lockTwTapPosition()` will prevent any message retry on exceptions

Severity: Low Risk

Context: BaseTapOFT.sol#L165-L169

Description: LayerZero documentation on message passing implies that it is up to the application on the destination chain to manage any message-handling triggered logic-/EVM-level failures and future retries of the same. This means that wherever the destination user application (UA) expects message failures it is responsible for storing them to enable future retries. While the protocol enforces this in most required places on destination chain functions, a call to `_storeFailedMessage()` is missing in the `catch Error` block of `BaseTapOFT._lockTwTapPosition()`.

Impact: Missing `_storeFailedMessage()` in `catch Error` block of `BaseTapOFT._lockTwTapPosition()` will prevent any message retry on exceptions.

Likelihood: Low + Impact: Medium = Severity: Low.

Recommendation: Add the missing `_storeFailedMessage` in `catch Error` block of `BaseTapOFT._lockTwTapPosition()`.

Tapioca: Addressed in PR 142.

5.4.22 Missing `ReceiveFromChain` event emissions in `BaseTapOFT` destination chain functions will cause mismatched events across chains

Severity: Low Risk

Context: BaseTapOFT.sol#L147-L182, BaseTapOFT.sol#L379-L439, BaseTapOFT.sol#L240-L326

Description: TOFT and USDO modules emit `SendToChain` events on the source chain and corresponding `ReceiveFromChain` events on the destination chain. These matched event emissions are typically used in cross-chain messaging to monitor and account for any dropped/missing messages, which may be used to retry messages depending on the specific protocol logic. However, while `BaseTapOFT` source functions `lockTwTapPosition()`, `unlockTwTapPosition()` and `claimRewards()` emit a `SendToChain` event, their corresponding destination chain functions are missing a `ReceiveFromChain` event emission.

Impact: This will cause mismatched events and may affect any monitoring/accounting tools/logic relying on these events.

Likelihood: High + Impact: Very Low = Severity: Low.

Recommendation: Consider adding `ReceiveFromChain` emissions in `BaseTapOFT` destination chain functions.

Tapioca: Addressed in PR 141.

5.4.23 Wrong singularity can be deleted with `unregisterSingularity()`, removing TAP emission from it

Severity: Low Risk

Context: TapiocaOptionLiquidityProvision.sol#L342-L354

Description: Instead of reverting `unregisterSingularity()` can end up deleting the last asset even if it's the wrong singularity (see TapiocaOptionLiquidityProvision.sol#L329-L346):

```
function unregisterSingularity(
    IERC20 singularity
) external onlyOwner updateTotalSGLPoolWeights {
    uint256 sglAssetID = activeSingularities[singularity].sglAssetID;
    if (sglAssetID == 0) revert NotRegistered();

    unchecked {
        uint256[] memory _singularities = singularities;
        uint256 sglLength = _singularities.length;
        uint256 sglLastIndex = sglLength - 1;

        for (uint256 i; i < sglLength; i++) {
            // If last element, just pop
            // see the line below
            if (i == sglLastIndex) {
                delete activeSingularities[singularity];
                delete sglAssetIDToAddress[sglAssetID];
                singularities.pop();
            }
```


I.e. if it is `_singularities[sglLastIndex] != sglAssetID` for any reason, the function will execute successfully and delete the wrong `(singularity, asset)` that happened to be the last one. This can go unnoticed since `UnregisterSingularity` doesn't emit `_singularities` element that was deleted.

Impact: incorrect singularity can be deleted, conditional on the one that was requested being in fact missed in `_singularities`, resulting in corrupted TapiocaOptionLiquidityProvision configuration, which will not emit rewards to the removed singularity as `_emitToGauges()` rely on `getSingularityPools()` and `_singularities` array (see TapiocaOptionBroker.sol#L707-L708):

```
function _emitToGauges(uint256 _epochTAP) internal {
    SingularityPool[] memory sglPools = tOLP.getSingularityPools();
```

Likelihood: very Low + Impact: High = Severity: Low.

Recommendation: Given that the function is used rarely and the risk described doesn't look to worth a minor gas savings, consider simplifying the logic and checking for `_singularities[i] == sglAssetID` all the time, e.g. TapiocaOptionLiquidityProvision.sol#L335-L359:

```
  unchecked {
      uint256[] memory _singularities = singularities;
      uint256 sglLength = _singularities.length;
      uint256 sglLastIndex = sglLength - 1;

      for (uint256 i; i < sglLength; i++) {
-         // If last element, just pop
-         if (i == sglLastIndex) {
-             delete activeSingularities[singularity];
-             delete sglAssetIDToAddress[sglAssetID];
-             singularities.pop();
-         } else if (_singularities[i] == sglAssetID) {
+         if (_singularities[i] == sglAssetID) {
              // If in the middle, copy last element on deleted element, then pop
              delete activeSingularities[singularity];
              delete sglAssetIDToAddress[sglAssetID];

-             singularities[i] = _singularities[sglLastIndex];
+             if (i != sglLastIndex) singularities[i] = _singularities[sglLastIndex];
              singularities.pop();
+             emit UnregisterSingularity(address(singularity), sglAssetID);
              break;
          }
      }
  }

-  emit UnregisterSingularity(address(singularity), sglAssetID);
  }
```

Tapioca: Fixed in PR 140.

Spearbit: Fix looks ok.

5.4.24 `BaseTapOFT.lockTwTapPosition()` incorrectly logging zero amount may lead to mismatched accounting

Severity: Low Risk

Context: BaseTapOFT.sol#L139-L144, BaseTapOFT.sol#L122

Description: `BaseTapOFT.lockTwTapPosition()` allows a TapOFT holder to `twTap.participate()` cross-chain by sending TapOFT `amount`. While the TapOFT `amount` sent is debited from `msg.sender`, the subsequent `SendToChain` event emitted incorrectly logs the sent amount as zero.

Impact: If these events are used for cross-layer bookkeeping, they will lead to mismatched accounting.

Likelihood: High + Impact: Very Low (depends on how the offchain tooling/accounting uses this event) = Severity: Low.

Recommendation: Use `amount` instead of `0` in `SendToChain` event emission.

Tapioca: Addressed in PR 138.

5.4.25 Accidentally changing the `governanceChainIdentifier` will break TAP DSO emissions

Severity: Low Risk

Context: TapOFT.sol#L140-L149, TapOFT.sol#L217-L218, TapOFT.sol#L156-L166

Description: TapOFT has a notion of `governanceChainIdentifier` set at initialization which identifies the chain's EVM identifier where TAP issuance distribution is minted for different categories and is later emitted for TAP DSO incentives. `TapOFT.emitForWeek()` updates weekly emission only on this governance chain. However, there is a provided setter `setGovernanceChainIdentifier()` which allows the owner to change `governanceChainIdentifier` at any point.

Impact: Accidentally changing the `governanceChainIdentifier` will break TAP DSO emission schedule on the original governance chain.

Likelihood: Very low + Impact: Medium = Severity: Low.

Recommendation: Reconsider whether this setter `setGovernanceChainIdentifier()` is really required and if so evaluate how the emission schedule can be ported over from the previous governance chain to the new one. If not, consider the implications of removing this setter.

Tapioca: Confirming this. it's already been removed on the v2 migration.

5.4.26 Use of different pause management functionality across protocol may cause unexpected behavior

Severity: Low Risk

Context: TapOFT.sol#L170-L174, Market.sol#L280-L285, BaseUSDO.sol#L176-L180, Penrose.sol#L291-L296, twTAP.sol#L9, TapiocaOptionBroker.sol#L9, TapiocaOptionLiquidityProvision.sol#L8, AirdropBroker.sol#L10

Description: Different contracts across the protocol use different pause management functionality via OpenZeppelin's Pausable or self-implemented pause logic.

Impact: Use of different pause management functionality across protocol may cause unexpected behavior due to incorrect/inconsistent application of pause-related modifiers, mismatched expectations of checks/modes or event emissions, which may lead to slower/incorrect incident response during emergency situations.

Likelihood: Low + Impact: Medium = Severity: Low.

Recommendation: Consider using the same pause management functionality across the protocol, wherever possible, for consistent expectation/behavior and preferably the widely used Pausable.sol.

Tapioca: Acknowledged. We'll keep the current implementation to avoid any extra size added to the contracts. Also, for markets we need the pause per operation

Spearbit: Acknowledged.

5.4.27 `Balancer.initConnectedOFT()` allows re-initializing MetaTOFTs which may prevent future rebalancings

Severity: Low Risk

Context: Balancer.sol#L245-L270, Balancer.sol#L196

Description: `Balancer.initConnectedOFT()` allows the owner to register a MetaTOFT configuration `connectedOFTs[_srcOft][_dstChainId]` and initialize it with `OFTData` for rebalancing. However, there are no checks to prevent accidental reinitializations. This allows resetting the value of `rebalanceable` to zero, which will prevent future calls to `rebalance()`.

Impact: `Balancer.initConnectedOFT()` allows owner to accidentally re-initialize MetaTOFTs which may prevent future rebalancings.

Likelihood: Low (requires owner to accidentally reinitialize) + Impact: Medium (assuming owner can detect and restore it to a previously valid state) = Severity: Low.

Recommendation: Add a precautionary check in the beginning of `Balancer.initConnectedOFT()` to validate if the particular configuration has already been initialized, for e.g., by checking if `connectedOFTs[_srcOft][_dstChainId].rebalanceable > 0`.

Tapioca: Addressed in PR 148.

5.4.28 `Balancer.onlyValidSlippage()` allowing up to 100% slippage may result in unexpectedly lower minimum output amounts from router swap

Severity: Low Risk

Context: Balancer.sol#L124-L127, Balancer.sol#L183-L193, Balancer.sol#L404-L420

Description: `Balancer.onlyValidSlippage()` allows a valid slippage of up to 100% with this upper threshold check: `if (_slippage >= 1e5) revert SlippageNotValid()`. This allows the owner to accidentally specify a very high unreasonable value of slippage during `rebalance()`.

Impact: This may allow unexpectedly lower minimum output amounts from router swap during rebalancing.

Likelihood: Low (depends on high value accidentally passed by owner) + Impact: Medium (excessive slippage) = Severity: Low.

Recommendation: Reconsider checking upper threshold of valid slippage against something smaller and more reasonable in `Balancer.onlyValidSlippage()`.

Tapioca: Addressed in PR 147.

5.4.29 Fees harvestable TOFTs are tracked but there is no function to harvest any fees if collected

Severity: Low Risk

Context: TapiocaWrapper.sol#L24-L25, TapiocaWrapper.sol#L155-L157, TapiocaWrapper.sol#L34-L35, TapiocaWrapper.sol#L44-L45, Tapioca Documentation, TapiocaWrapper.sol#L96-L102

Description: While the implementation tracks TOFTs deployed on the host chain as "fees harvestable" TOFTs, the documentation references a `harvestFees()` function which was present earlier but appears to have been removed since then. The current implementation appears to also have a stale error and event related to setting of management fee. It is therefore not clear if the current specification expects fees to be applied on TOFT utlization because other parts of the documentation refer to this being zero fee.

Impact: If the specification expects TOFT fees, this functionality is missing and will result in loss to the protocol. If not, the documentation is not consistent and the tracking of `harvestableTapiocaOFTs` is unnecessary.

Likelihood: Medium + Impact: Low (assuming no fees is expected) = Severity: Low.

Recommendation: Revisit the specification for TOFT fees to either introduce fee collection+harvesting or make the documentation consistent with the implementation by removing the unnecessary tracking logic.

Tapioca: Addressed in PR 145.

5.4.30 Overfunding a batch call does not refund the excess ETH balance

Severity: Low Risk

Context: TapiocaWrapper.sol#L105-L129

Description: `executeCalls()` allows the owner to batch execute different functions on different TOFTs. Because many functions require ETH for different reasons, `executeCalls()` is payable and passes the specified `_call[i].value` to the calls while tracking the accumulated ETH sent in `valAccumulator`. However, if the owner accidentally passes more `msg.value` than the sum total of all ETH required for the batched calls, the remainder balance is not refunded back but is left in the wrapper contract.

Impact: Accidentally overfunding a batch call does not refund the excess balance back to the owner.

Likelihood: Low + Impact: Medium (excess balance is lost) = Severity: Low.

Recommendation: Consider refunding the excess `totalVal - valAccumulator` back to the owner.

Tapioca: Addressed in PR 144.

5.4.31 `Singularity` interest rate may be gameable by whales for some profit

Severity: Low Risk

Context: SGLCommon.sol#L123-L148

Description: `_getInterestRate` computes interest rates by following the logic:

- If we're below `minimumTargetUtilization` we reduce the interest rate down towards the minimum value.
- If we're above `maximumTargetUtilization` we increase the interest rate down towards the maximum value.
- If utilization is between the two, we do nothing.

Due to this, there may be scenarios in which a whale (has provided a lot of asset to borrow) could purposefully borrow some marginal amount for a very short period of time, as a means to nudge the interest in their favor.

The proof of concept below shows the feasibility of this, we recommend further economic research is done as it should be possible to raise interest in a way that is profitable to the lender even in spite of them having to pay a borrow fee to cause all interest rates to increase.

The following proof of concept shows how a raise of 20 BPS of borrowed amount can be used to increase borrow rates by 1 wei, over 12 seconds. Longer periods and higher amounts can be used to raise the rate further.

Provided that a whale has a sufficiently high ownership of the supplied asset, the increased APR of borrowing fees will compensate for the cost the whale has to pay to raise the rate.

Since borrow rates only change in drastic "underborrowing" and "overborrowing" it may take a while for rates to reduce after they have been nudged towards a certain direction

Proof of Concept:

```
function testNudgeUp() public {
    target.setMinInterest(158548960);
    target.setMaxInterest(317097920000);
    target.setMinTargetUtilization(3e17); // 10%
    target.setMaxTargetUtilization(5e17); // 80%

    target.setStartInterestPerSecond(158548960);

    console2.log("currentInterestPerSecond", target.currentInterestPerSecond());

    // let's try nudging up
    uint256 increaseAmt = 2e15; // 20 BPS
    target.updateIntestRate(5e17 + increaseAmt,  12); // TODO: After what amount is this profitable to
↪   do?
    console2.log("currentInterestPerSecond", target.currentInterestPerSecond());
}
```

Which will show

```
[PASS] testNudgeUp() (gas: 128942)
Logs:
  currentInterestPerSecond 158548960
  currentInterestPerSecond 158548961
```

A full proof of concept is available here, and can be run via `forge test --match-test testNudgeUp -vv`.

Tapioca: Acknowledged.

Spearbit: Acknowledged.

5.4.32 `tapSendData.amount` is ignored in `BaseTOFTOptionsDestinationModule.exercise()` to send the entire `tapAmount` to `lzDstChainId` chain

Severity: Low Risk

Context: BaseTOFTOptionsDestinationModule.sol#L180, ITapiocaOptionsBroker.sol#L21-L28

Description: `BaseTOFTOptionsModule.exerciseOption()` allows a user to specify `ITapiocaOptionsBrokerCrossChain.IExerciseLZSendTapData` which includes options for sending the exercised TAP to another chain if `withdrawOnAnotherChain == true`. This includes an `amount` which presumably is the amount of exercised TAP to be sent to the specified `lzDstChainId` chain. However, `BaseTOFTOptionsDestinationModule.exerciseInternal()` sends the entire `tapAmount` to `lzDstChainId` chain instead of sending only `amount` over there and then sending the remaining `tapAmount - tapSendData.amount`, if any, to the user on the executed destination chain.

Impact: User will unexpectedly receive the entire `tapAmount` on `lzDstChainId` chain if `withdrawOnAnotherChain == true` instead of only `amount`, requiring them to again transfer back TAP to the executed destination chain for their intended reason.

Likelihood: Medium (requires `withdrawOnAnotherChain == true`) + Impact: Low = Severity: Low.

Recommendation: Add logic in `BaseTOFTOptionsDestinationModule.exerciseInternal()` to `sendFrom` only `tapSendData.amount` and then `IERC20(tapSendData.tapOftAddress).safeTransfer(from, tapAmount-tapSendData.amount)` any remaining balance.

Tapioca: Created PR 138.

5.4.33 Missing `ReceiveFromChain` emission in `BaseTOFTMarketDestinationModule.remove()` will cause mismatched events across chains

Severity: Low Risk

Context: BaseTOFTMarketModule.sol#L104, BaseTOFTMarketDestinationModule.sol#L176-L256

Description: All TOFT modules emit `SendToChain` events on the source chain and corresponding `ReceiveFromChain` events on the destination chain. These matched event emissions are typically used in cross-chain messaging to monitor and account for any dropped/missing messages, which may be used to retry messages depending on the specific protocol logic. However, while `BaseTOFTMarketModule.removeCollateral()` emits a `SendToChain` event, the corresponding `BaseTOFTMarketDestinationModule.remove()` is missing a `ReceiveFromChain` event emission.

Impact: This will cause mismatched events and may affect any monitoring/accounting tools/logic relying on these events.

Likelihood: High + Impact: Very Low = Severity: Low.

Recommendation: Add `ReceiveFromChain` emission in `BaseTOFTMarketDestinationModule.remove()`

Tapioca: Created PR 137.

5.4.34 Anyone can donate to a native TOFT vault to make it report an artificially inflated supply

Severity: Low Risk

Context: TOFTVault.sol#L44, TOFTVault.sol#L23-L26

Description: Every TOFT, which is minted 1:1 for underlying ERC20 or ETH if it's a native TOFT, is associated with a vault for its ERC20 tokens or ETH. A native TOFT vault's `viewSupply()` uses `address(this).balance` for reporting its TOFT supply. However, the vault contract has a `receive() external payable {}`, which is unnecessary because a vault is only expected to receive ETH via `depositNative()` when someone mints its corresponding native TOFT via `wrap()` or during rebalancing via `sgReceive()`.

Impact: Anyone can donate to a native TOFT vault to make it report an artificially inflated TOFT supply, which may affect Tapioca integrations that use the vault's `viewSupply()` getter.

Likelihood: Low + Impact: Low = Severity: Low.

Recommendation: Remove the unnecessary `receive()` function from `TOFTVault.sol`.

Tapioca: Created PR 134.

5.4.35 Missing events in privileged functions may cause unexpected protocol changes for users

Severity: Low Risk

Context: BaseTOFTStorage.sol#L93-L95, BaseTOFT.sol#L532-L535, BaseTOFT.sol#L537-L539, Balancer.sol#L172-L174, Balancer.sol#L228-L238, Balancer.sol#L290-L346, AirdropBroker.sol#L352-L356, AirdropBroker.sol#L358-L374

Description: There are many privileged functions in the protocol which update critical protocol addresses or parameters affecting all users, but do not emit an event for offchain monitoring/tooling to observe and react appropriately.

Impact: This may cause sudden and unexpected protocol changes for all users preventing them from appropriately managing their protocol interactions and any risk mitigations.

Likelihood: Medium + Impact: Low = Severity: Low.

Recommendation: Add appropriate event emissions in all privileged functions.

Tapioca: Created PR 133.

5.4.36 Airdropped options cannot be exercised with tokens that have greater than 18 decimals

Severity: Low Risk

Context: AirdropBroker.sol#L563-L577, TapiocaOptionBroker.sol#L694-L702

Description: Users are allowed to exercise their options by paying in whitelisted ERC20 tokens. While most tokens have less than 18 decimals, the protocol appears to allow the use of payment tokens with greater than 18 decimals as illustrated by the supported logic in TapiocaOptionBroker.sol#L694-L702. However, the similar logic for exercising airdropped options is missing support for tokens with greater than 18 decimals.

This was also reported in C4-879.

Impact: Airdropped options cannot be exercised with tokens that have greater than 18 decimals. Users will have to use other payment tokens.

Likelihood: Low + Impact: Low = Severity: Low.

Recommendation: Add support in `AirdropBroker._getDiscountedPaymentAmount()` for airdropped options to be exercised with tokens that have greater than 18 decimals, similar to that in `TapiocaOptionBroker`:

```
- paymentAmount = paymentAmount / (10 ** (18 - _paymentTokenDecimals));
+ paymentAmount = paymentAmount / (10 ** (_paymentTokenDecimals > 18 ? (10 ** (_paymentTokenDecimals -
↪  18)) : (10 ** (18 - _paymentTokenDecimals)))
```

Tapioca: Fixed in PR 134.

5.4.37 Last withdrawer will be unable to withdraw all of their asset from `Singularity`

Severity: Low Risk

Context: SGLCommon.sol#L252-L254

Description: Singularity alleviates rebase attacks by enforcing a minimum `base` of 1000 units (see SGLCommon.sol#L252-L254):

```
_totalAsset.base -= uint128(fraction);
if (_totalAsset.base < 1000) revert MinLimit();
```

That same check, in the context of `removeAsset` will cause the last withdrawer to be unable to withdraw those remaining 1000 weis. This is generally not a big risk, as the economic value lost is low.

However, it's worth adding this to the documentation for 2 reasons:

1. The last person to withdraw will have to forfeit those 1000 weis.
2. Any automated piece of code, that may attempt to `removeAsset(balanceOf(address(this))` will revert until another deposit of at least 1000 wei is done .

Recommendation: Document the behaviour in comments and in the developer docs.

Tapioca: Acknowledged.

Spearbit: Acknowledged.

5.4.38 `setUsdoToken()` should be restricted to a single use

Severity: Low Risk

Context: Penrose.sol#L511

Description: The `Penrose` contract defines a canonical USDO token contract address, `usdoToken`, but includes the function `setUsdoToken` that can change this state variable. The BigBang and Singularity market contracts are aware of the canonical USDO token address only at their construction time when their `init` functions are called. Any change to `usdoToken` will leave the market outdated and in need of replacement. It is unclear how the deployed market contracts, borrowers, and other end users will be expected to adapt to a change of the canonical token address, and it seems this function is intended to be restricted to a single use.

Recommendation: Restrict the `setUsdoToken` function to a single use, or remove the function and move the setter logic of `usdoToken` and `usdoAssetId` in the constructor.

Tapioca: Created PR 294.

Spearbit: Verified.

5.4.39 Honorary Pearl Club NFT holders may claim in Phase 3 of Tapioca Option Airdrop breaking eligibility criterion and exceeding Phase 3 allocation

Severity: Low Risk

Context: AirdropBroker.sol#L474-L495, Pearl Club NFT Collection, Tapioca Option Airdrop Phase 3

Description: Pearl Club NFT Collection says:

> Pearl Club ONFTs 001 through 700 were distributed to eligible Guild members who beta tested the Tapioca platform and provided feedback. 769 to 783 were given to core contributors of TapiocaDAO, and are honorary 1-of-1 Pearl Club NFTs.

and Tapioca Option Airdrop Phase 3 says:

500,000 TAP will be allocated to the Pearl Club NFT holders in the form of aoTAP call options with a 48-hour expiry. There are 700 eligible Pearl Club NFTs (honorary PCNFT's are not included), thus each bearing 714 aoTAP.

But the check to prevent the 14 "honorary PCNFT" holders (tokenIDs 769-783) from claiming in Phase 3 is not enforced. Besides breaking the eligibility criterion, allowing honorary PCNFT holders to claim here will make Phase 3 claims to either exceed its 500,000 TAP allocation or unable to airdrop 714 aoTAP to all 700 eligible non-honorary PCNFT holders.

Likelihood: Low + Impact: Medium = Severity: Low.

Recommendation: The missing check `_tokenID > 0 && _tokenID <= 700` should be added to `_participatePhase3()`.

Tapioca: Created PR 133.

5.4.40 `aoTAP.mint()`, `oTap.mint` and `TapiocaOptionLiquidityProvision.lock` are susceptible to reentrancy

Severity: Low Risk

Context: aoTAP.sol#L120, AirdropBroker.sol#L438, oTAP.sol#L118-L126, TapiocaOptionLiquidityProvision.sol#L215-L217

Description: Multiple functions in `TapiocaZ` such as `aoTAP.mint()`, call `_safeMint()` executing a callback on the destination contract's `onERC721Received` which can be used to reenter. This function is called by `AirdropBroker` during the mint for all four participation phases, e.g. `_participatePhase1()`.

Impact: This flow is susceptible to reentrancy and CEI is not followed here given the subsequent state change effects for creating the option position. But it does not appear to be obviously exploitable.

Likelihood: Low + Impact: Low = Severity: Low.

Description: Consider replacing `_safeMint` with `_mint` or moving `_safeMint` to the interactions part of the code for enforcing CEI pattern.

Tapioca: Was fixed independently during our v2 migration, in PR 151.

Spearbit: Verified.

5.4.41 `TapiocaOptionLiquidityProvision` inherits `Pausable` but functions do not use its modifier

Severity: Low Risk

Context: TapiocaOptionLiquidityProvision.sol#L8, TapiocaOptionLiquidityProvision.sol#L52, TapiocaOptionLiquidityProvision.sol#L196-L201, TapiocaOptionLiquidityProvision.sol#L233-L237

Description: `TapiocaOptionLiquidityProvision` inherits `Pausable` but its `lock()` and `unlock()` state-modifying functions do not use its `whenNotPaused` modifier to enable pausing capability for any emergency.

Impact: `TapiocaOptionLiquidityProvision` functions cannot be paused during any emergency.

Likelihood: Low + Impact: Low = Severity: Low.

Recommendation: Add Pausable modifiers to `TapiocaOptionLiquidityProvision` `lock()` and `unlock()` functions.

Tapioca: Created PR 132.

5.4.42 `emitForWeek()` may be called even when `TapOFT` is paused

Severity: Low Risk

Context: TapOFT.sol#L217, TapOFT.sol#L255, TapOFT.sol#L268

Description: State-modifying functions `extractTAP()` and `removeTAP()` in `TapOFT` enforce the `notPaused` modifier to prevent state updates when paused for any emergency. However, `emitForWeek()` is missing this modifier.

Impact: `emitForWeek()` may be called by `TapiocaOptionBroker.newEpoch()` even when `TapOFT` is paused causing unexpected state updates.

Looks like `notPaused` was initially present but was removed as a recommendation from C4-1218.

Likelihood: Low + Impact: Low = Severity: Low.

Recommendation: Reconsider adding `notPaused` modifier to `emitForWeek()` if it is better to pause and lose emission or instead keep this going even when paused.

5.4.43 Most `USDO` Destination Functions cannot be retried due to the `(msg.sender != address(this))` check

Severity: Low Risk

Context: USDOLeverageDestinationModule.sol#L32-L33)

Description: All Destination Modules functions have the following check:

- USDOLeverageDestinationModule.sol#L32-L33

```
if (msg.sender != address(this)) revert SenderNotAuthorized();
```

This is enforcing that the call is being performed by self.

When using a `NonblockingLzApp`, the LayerZero relayer will call: `_blockingLzReceive`

- NonblockingLzApp.sol#L24-L30

```
function _blockingLzReceive(uint16 _srcChainId, bytes memory _srcAddress, uint64 _nonce, bytes
↪  memory _payload) internal virtual override {
    (bool success, bytes memory reason) = address(this).excessivelySafeCall(gasleft(), 150,
↪  abi.encodeWithSelector(this.nonblockingLzReceive.selector, _srcChainId, _srcAddress, _nonce,
↪  _payload));
    // try-catch all errors/exceptions | /// @audit we need `reason` mloading to cause usage in
↪  excess of 1/64 or max gas
    if (!success) { /// 468750 gas, seems like we cannot break it then
        _storeFailedMessage(_srcChainId, _srcAddress, _nonce, _payload, reason);
    }
}
```

Which will in turn use `ExcessivelySafeCall` to self. This will make the check above pass, since `msg.sender == address(this)`.

However, when calling `retryMessage` the `NonblockingLzApp` will directly call `_nonblockingLzReceive`, which will skip calling self.

- NonblockingLzApp.sol#L46-L56

```
function retryMessage(uint16 _srcChainId, bytes calldata _srcAddress, uint64 _nonce, bytes
↪  calldata _payload) public payable virtual {
    // assert there is message to retry
    bytes32 payloadHash = failedMessages[_srcChainId][_srcAddress][_nonce];
    require(payloadHash != bytes32(0), "NonblockingLzApp: no stored message");
    require(keccak256(_payload) == payloadHash, "NonblockingLzApp: invalid payload");
    // clear the stored message
    failedMessages[_srcChainId][_srcAddress][_nonce] = bytes32(0);
    // execute the message. revert if it fails again
    _nonblockingLzReceive(_srcChainId, _srcAddress, _nonce, _payload);
    emit RetryMessageSuccess(_srcChainId, _srcAddress, _nonce, payloadHash);
}
```

Meaning that the `msg.sender` will be the caller. For this reason, the `DestinationModule` functions will always revert when called via `retryMessage`.

Recommendation: Consider fully documenting the behaviour around which functions should work when calling `retryMessage`. From reviewing the codebase, not being able to retry can be considered intended, however, it should be fully documented as it could be used to prevent certain user operations from being executed.

List of Instances:

- `leverageUp`.
- `lend`.
- `remove`.
- `exercise`.

Tapioca: PRs created: 295, 130.

5.4.44 Accidentally sent user ETH will get locked in a non-native TOFT

Severity: Low Risk

Context: TapiocaOFT.sol#L69-L74, mTapiocaOFT.sol#L111-L117

Description: Users who accidentally also sent ETH along with a TOFT's underlying ERC20 during wrapping will have their ETH locked in the TOFT contract.

Impact: Accidentally sent user ETH will get locked in a non-native TOFT

Likelihood: Low + Impact: Medium = Severity: Low.

Recommendation: Consider adding a `msg.value == 0` check in the `else{}` block.

Tapioca: Created PR 127.

5.4.45 Anyone is allowed to call `sgReceive()` if `_stargateRouter` is not set

Severity: Low Risk

Context: BaseTOFT.sol#L499-L509, BaseTOFT.sol#L537-L539

Description: The Stargate router address in TOFT contracts is not set by default or at initialization but is expected to be set by the owner using the setter `setStargateRouter()`. The `sgReceive()` function which is expected (from code/comments) to always be callable only by Stargate router enforces `msg.sender == _stargateRouter` check only if `_stargateRouter != address(0)`.

Impact: An attacker can front-run the call to setter `setStargateRouter()` and successfully call `sgReceive()` which will be unexpected. This will transfer any underlying TOFT ETH/ERC20, if present, to its vault.

Likelihood: Low + Impact: Low = Severity: Low.

Recommendation: Remove the conditional check `_stargateRouter != address(0)` to always enforce the authorization. Consider setting `_stargateRouter` in the constructor:

```
- if (_stargateRouter != address(0)) {
  if (msg.sender != _stargateRouter) revert NotAuthorized();
- }
```

Tapioca: Created PR 128.

5.4.46 Use of different ownership management libraries across protocol may cause unexpected behavior

Severity: Low Risk

Context: Balancer.sol#L10, TapiocaWrapper.sol#L9, twTAP.sol#L6, Ownable.sol, BoringOwnable.sol, Owned.sol

Description: Different contracts across the protocol use different ownership management libraries of OpenZeppelin's `Ownable`, Boringcrypto's `BoringOwnable` and Solmate's `Owned`. These libraries allow different modes of and checks for ownership transfer: `Ownable` provides a single-step transfer with a zero-address check along with a `renounceOwnership()` functionality; `BoringOwnable` provides an optional single-step/direct transfer with a zero-address check or a two-step ownership transfer; `Owned` provides a single-step transfer with no zero-address check.

Impact: Use of different ownership management libraries across protocol may cause unexpected behavior due to incorrect expectations of checks/modes during ownership transfers, which may lead to accidental ownership renunciations or incorrect transfers.

Likelihood: Low + Impact: Medium = Severity: Low.

Recommendation: Consider using the same ownership management library across the protocol for consistent Expectation/behavior and preferably one with a two-step transfer with appropriate checks such as Ownable2Step.sol.

Tapioca: Addressed in PR 129.

5.4.47 Functions `advanceWeek()` and `distributeReward()` may be called even when `twTAP` is paused

Severity: Low Risk

Context: twTAP.sol#L442, twTAP.sol#L473-L476

Description: All state-modifying functions in `twTAP` enforce the `whenNotPaused` modifier to prevent state updates when paused for any emergency. However, `advanceWeek()` and `distributeReward()` are missing this modifier.

Impact: Functions `advanceWeek()` and `distributeReward()` may be called even when `twTAP` is paused causing unexpected state updates.

Likelihood: Low + Impact: Low = Severity: Low.

Recommendation: Consider adding `whenNotPaused` modifier to both functions.

Tapioca: Fixed in PR 129.

5.4.48 Funds holding singularity can be unregistered, freezing the assets

Severity: Low Risk

Context: TapiocaOptionLiquidityProvision.sol#L329-L359

Description: It is now possible to unregister a singularity with `activeSingularities[_singularity].totalDeposited > 0`.

It will make unlocking impossible as `unlock()` reverts when non-empty `lockPosition.sglAssetID` recorded doesn't correspond to the erased `activeSingularities[_singularity].sglAssetID`:

- TapiocaOptionLiquidityProvision.sol#L251-L252:

```
if (sgl.sglAssetID != lockPosition.sglAssetID)
    revert InvalidSingularity();
```

Impact: when deposit holding singularity being unregistered via `unregisterSingularity()` the corresponding funds will be permanently frozen.

Per very low likelihood and high principal funds loss impact setting the severity to be low.

Recommendation: Consider the check for `activeSingularities[_singularity].totalDeposited > 0` is recommended along with an additional flag for `unregisterSingularity()` that allows for overriding it for the case of some dust or long-forgotten locks interfering with the cleanup.

I.e. normally `unregisterSingularity()` should revert on `totalDeposited > 0`, but with the flag the current unconditional behavior can be forced. Rescue mode can serve as such a flag to streamline the logic and reduce the overall centralization surface.

Tapioca: Since there's not a 100% chance of having totalDeposited == 0, we want to still be able to unregister in that case. The logic would be to enter in rescue mode so that users can unlock, and wait for a certain time before doing so. I guess a check for rescue mode could be added.

Fixed in PR 127.

I think we can go with the bool value, but I added a cooldown period to activate. Since there could be 2 different type of scenarios that could happen, either we want it to be fast and so the cooldown is 0, and we activate rescue ASAP, or we roll with the default of 2 days or whatever the cooldown is gonna be by then.

Fixed in PR 160.

Spearbit: PR 127 fix only requires for the singularity being deleted to be in rescue mode. Consider adding the time of rescue mode triggering (e.g. timestamp instead of the current `bool` value) and requiring some predetermined period to be passed since rescue was triggered in order to unregister.

For PR 160 note that cooldown on rescue trigger itself, being an improvement, will not change much for locked depositors as they can exit only when rescue mode is already triggered as `lockDuration` cannot be surpassed before that. New trigger cooldown logic looks ok.

5.4.49 Vesting initialization can be griefed by donating tokens

Severity: Low Risk

Context: Vesting.sol#L217

Description: After registering users, the Vesting owner is expected to initialize the process by calling `init()` and specifying among other things the total vested amount `_seededAmount`. This `_seededAmount` is sanity checked against values of zero, `__totalAmount` and finally the available contract balance of tokens. However, the final check is implemented as a strict inequality check: `if (availableToken != _seededAmount) revert BalanceTooLow()`.

Impact: An attacker can grief this initialization process by donating a few tokens to this contract such that its balance is greater than `_seededAmount`, i.e. what the owner expects it to be after registering users. This will cause `init()` to revert and require owner to initialize again with the correct balance amount while avoiding repeated griefing via front-running.

Likelihood: Low + Impact: Low = Severity: Low.

Recommendation: Change:

```
- if (availableToken != _seededAmount) revert BalanceTooLow();
+ if (availableToken < _seededAmount) revert BalanceTooLow();
```

Tapioca: Fixed in PR 125.

5.4.50 Incorrect threshold check in `TapiocaOptionLiquidityProvision.lock()` may lead to user loss of oTAP DSO incentives

Severity: Low Risk

Context: TapiocaOptionLiquidityProvision.sol#L202, TapiocaOptionBroker.sol#L94, Documentation

Description: `TapiocaOptionLiquidityProvision.lock()` performs a threshold check for `_lockDuration` parameter against 0. This should really be against `EPOCH_DURATION` as commented `// 7 days = 604800` and enforced in TapiocaOptionBroker.participate().

This is also supported by the docs:

> When a user locks their SGL receipt token and thus the underlying lending position, they can select any lock duration they wish, in units of epochs (weeks), with one epoch being the minimum escrow time,...

Impact: A lender locks their SGL position to mint tOLP accidentally for a duration less than `EPOCH_DURATION`. The minted tOLP does not allow them to mint an oTAP position because of the `EPOCH_DURATION` check enforced in `TapiocaOptionBroker.participate()`. They also cannot rectify their error by unlocking that tOLP at this point until the expiration of `lockPosition.lockDuration`. This leads to user loss of oTAP DSO incentives for the duration of the lock.

Likelihood: Low + Impact: Medium = Severity: Low.

Recommendation: Enforce a minimum threshold check of `EPOCH_DURATION` for `_lockDuration` in `TapiocaOptionLiquidityProvision.lock()`.

Tapioca: Fixed in PR 126.

5.4.51 `TapiocaOptionLiquidityProvision.lock()` is susceptible to reentrancy

Severity: Low Risk

Context: TapiocaOptionLiquidityProvision.sol#L217-L224

Description: `TapiocaOptionLiquidityProvision.lock()` calls `_safeMint()` executing a callback on the destination contract's `onERC721Received` which can be used to reenter.

Impact: This is susceptible to reentrancy and CEI is not followed here given the subsequent state change effects for creating the lock position. But it does not appear to be obviously exploitable.

Likelihood: Low + Impact: Low = Severity: Low

Recommendation: Add `nonReentrant` modifier as a precaution. Consider the possibility of enforcing CEI pattern to mitigate any cross-function reentrancy risk.

Tapioca: Fixed in PR 122.

5.4.52 `TapiocaOptionBroker.participate()` is susceptible to reentrancy

Severity: Low Risk

Context: TapiocaOptionBroker.sol#L363-L368, oTAP.sol#L126

Description: `TapiocaOptionBroker.participate` calls `oTAP.mint()` that in turn calls `_safeMint()` executing a callback on the destination contract's `onERC721Received` which can be used to reenter.

Impact: This is susceptible to reentrancy but does not appear to be obviously exploitable because CEI pattern is implemented.

Likelihood: Low + Impact: Low = Severity: Low

Recommendation: Add `nonReentrant` modifier as a precaution.

Tapioca: Fixed in PR 121.

5.4.53 USDO and TOFT functions logging amounts with dust included may lead to mismatched accounting

Severity: Low Risk

Context: USDOOptionsModule.sol#L105-L110, USDOOptionsModule.sol#L65-L73, BaseTOFTOptionsModule.sol#L122, BaseTOFTStrategyDestinationModule.sol#L162

Description: Instead of logging `paymentTokenAmount` which has dust removed, `USDOOptionsModule.exerciseOption()` logs `optionsData.paymentTokenAmount` in the `SendToChain` event emission. This would have been avoided if the processing was done in-place using `optionsData.paymentTokenAmount` instead of a temporary `paymentTokenAmount` variable.

Incorrect amounts are similarly emitted in functions `BaseTOFTOptionsModule.exerciseOption()` and `BaseTOFTStrategyDestinationModule.strategyWithdraw()`.

Impact: If these events are used for cross-layer bookkeeping, they will lead to mismatched accounting.

Likelihood: High + Impact: Very Low = Severity: Low.

Recommendation: In `USDOOptionsModule.exerciseOption()`, change to:

```
  emit SendToChain(
      lzData.lzDstChainId,
      optionsData.from,
      toAddress,
-     optionsData.paymentTokenAmount
+     paymentTokenAmount
  );
```

Revisit amounts logged to determine if they are indeed correct or if their processed/unprocessed variants should be logged instead.

Tapioca: Fixed in PR 286.

Spearbit: Similar issue referenced above in BaseTOFTOptionsModule.sol#L122, BaseTOFTStrategyDestinationModule.sol#L162 doesn't appear to be fixed in PR 286.

5.4.54 SGL liquidation always logs the amount of asset added back as zero

Severity: Low Risk

Context: SGLLiquidation.sol#L289-L294, SGLLiquidation.sol#L285, SGLLiquidation.sol#L262

Description: Instead of logging the amount of asset added back `returnedShare - feeShare - callerShare` as calculated on L285, `LogAddAsset` captures it as `extraShare - feeShare - callerShare` which is always zero because `feeShare = extraShare - callerShare` from L262.

Impact: Incorrect offchain accounting of assets that are added upon liquidation.

Likelihood: High + Impact: Very Low = Severity: Low.

Recommendation: Change to:

```
  emit LogAddAsset(
      address(this),
      address(this),
-     extraShare - feeShare - callerShare,
+     returnedShare - feeShare - callerShare,
      0
  );
```

Tapioca: Fixed in PR 281.

5.4.55 Using older versions of OpenZeppelin dependencies may be error-prone

Severity: Low Risk

Context: Tapioca-bar.package.json#L48, tap-token.package.json#L25, TapiocaZ.package.json#L32, Snyk report

Description: The protocol uses various OpenZeppelin libraries across contracts. However, the dependencies allow the use of older versions of OpenZeppelin libraries (`"@openzeppelin/contracts": "^4.8.2"` in `Tapioca-bar` and `tap-token` repositories and `"@openzeppelin/contracts": "^4.5.0"` in `TapiocaZ` repository) which have had known vulnerabilities fixed in newer versions.

Impact: Using older versions of libraries with known vulnerabilities may be error-prone.

Likelihood: Low + Impact: Medium = Severity: Low

Recommendation: Consider upgrading to the latest version `5.0.1`.

Tapioca: PRs 131 and 301 created. Updated OZ to 4.9.5 as later versions require some code changes or custom implementations.

5.4.56 `BigBang.execute()` is susceptible to reentrancy

Severity: Low Risk

Context: BigBang.sol#L219-L222, Singularity.sol#L236-L241

Description: While `Singularity.execute()` has a `nonReentrant` modifier, the equivalent `BigBang.execute()`, which similarly allows batched calls to BB functions, does not have that modifier and so allows reentrancy on any of its functions making external calls.

Impact: Given that some BB functions, e.g. `liquidate()` do not strictly follow CEI, make external calls and themselves do not have a reentrancy guard, reentrancy is possible via `execute()` but does not appear to be obviously exploitable.

Likelihood: Low + Impact: Low = Severity: Low

Recommendation: Add a reentrancy guard to `execute()` as a precaution similar to `Singularity.execute()`.

Tapioca: Fixed at commit 8939b2075.

5.4.57 SGL and BB `liquidate()` is susceptible to reentrancy

Severity: Low Risk

Context: SGLLiquidation.sol#L89-L94, SGLLiquidation.sol#L144, BBLiquidation.sol#L87-L92, BBLiquidation.sol#L142

Description: The `liquidate()` function ends up calling `_liquidatorReceiver.onCollateralReceiver()` on the `_liquidatorReceiver` addresses provided by the liquidator.

Impact: Given that the function does not strictly follow CEI and does not have a reentrancy guard, reentrancy is possible but does not appear to be obviously exploitable.

Likelihood: Low + Impact: Low = Severity: Low

Recommendation: Add a reentrancy guard to SGL and BB `liquidate()` as a precaution. Consider the possibility of enforcing the CEI pattern to prevent cross-function reentrancy.

Tapioca: Fixed in PR 278.

5.4.58 Missing threshold check and inconsistent relative check for `collateralizationRate` allows for invalid values to break core logic

Severity: Low Risk

Context: BigBang.sol#L184-L193, Market.sol#L256-L275

Description: While `setMarketConfig()` enforces a threshold check of `_collateralizationRate <= FEE_PRECISION`, and `_collateralizationRate <= liquidationCollateralizationRate` relative check, `_initCoreStorage()` is missing a similar check for `collateralizationRate` against `FEE_PRECISION` and enforces `_collateralizationRate < liquidationCollateralizationRate` (< instead of <=).

Impact: Missing upper bound threshold check against `FEE_PRECISION` and inconsistent relative check against `liquidationCollateralizationRate` allows for accidentally set invalid values for this critical protocol parameter to break core logic.

Likelihood: Very Low + Impact: High = Severity: Low

Recommendation: Add missing upper bound threshold check and make the relative check against `liquidationCollateralizationRate` consistent.

Tapioca: Fixed in PR 277.

5.4.59 `MarketLiquidatorReceiver`: Enforcing oracles as well as a fixed path will cause suboptimal liquidations, which can cause systemic risk to the system

Severity: Low Risk

Context: MarketLiquidatorReceiver.sol#L148-L150

Description: `MarketLiquidatorReceiver` determines the *fair* amountOut by querying the oracle and then adding slippage (see MarketLiquidatorReceiver.sol#L148-L150):

```
uint256 tokenOutAmount = (tokenInAmount * rate) /
    oracles[_tokenIn].precision;
return tokenOutAmount - ((tokenOutAmount * _slippage) / 10_000); //50 is 0.5%
```

While this is superficially correct, an oracle is subject to a Deviation Threshold, which I will call drift; drift is the difference between the fair price of the asset onChain and the price reported by the oracle. This drift can cause swaps with "intended slippage" to revert as the actual slippage (+ fees) against the Oracle Price is higher than the intended slippage.

In those instances, slippage will have to be set to: `drift + swap fee`, in order to accommodate for the oracle delay. Overall enforcing a swap path for liquidations is not ideal as most MEV actors will have their own preferred way of swap (and hedge) meaning that enforcing a specific path will cause them to have tighter margins.

Higher slippage may instead open up the Liquidator to MEV, which would reduce their profitability, in turn making them less likely to perform Liquidations.

Recommendation: Consider whether it's an acceptable risk to enforce a specific liquidation swap path, or if you can refactor to allow the liquidator to simply repay the debt and then perform the swap externally. An external swap will lower the surface area of the system.

At the same time, removing privileged access to liquidations may cause a loss of yield to the DAO as sophisticated actors may be faster than the DAO without any privileged access.

Tapioca: PR created (see PR 293).

5.5 Gas Optimization

5.5.1 `_emitToGauges` will emit zero amounts to singularities in rescue mode, `_depositFeesToTwTap()` atomically deposits and withdraws the same funds

Severity: Gas Optimization

Context: TapiocaOptionLiquidityProvision.sol#L141-L159, TapiocaOptionBroker.sol#L707-L723, Penrose.sol#L511-L528, BigBang.sol#L485-L503

Description: When a particular SGL is in rescue mode `_emitToGauges()` will set `singularityGauges[epoch][sglAssetID] = quotaPerSingularity = 0` to `sglAssetID == address(0)` as `tOLP.getSingularityPools()` returns empty elements for such singularities (see TapiocaOptionLiquidityProvision.sol#L141-L159):

```
function getSingularityPools()
    external
    view
    returns (SingularityPool[] memory)
{
    // ...
    unchecked {
        // see the line below
        for (uint256 i; i < len; ++i) {
            SingularityPool memory sgl = activeSingularities[
                sglAssetIDToAddress[_singularities[i]]
            ];
            // If the pool is in rescue, don't return it
            if (sgl.rescue) {
                // see the line below
                continue;
            }
            pools[i] = sgl;
```


Also TapiocaOptionBroker.sol#L707-L723:

```
function _emitToGauges(uint256 _epochTAP) internal {
    // see the line below
    SingularityPool[] memory sglPools = tOLP.getSingularityPools();
    uint256 totalWeights = tOLP.totalSingularityPoolWeights();

    uint256 len = sglPools.length;
    unchecked {
        // For each pool
        for (uint256 i; i < len; ++i) {
            uint256 currentPoolWeight = sglPools[i].poolWeight;
            uint256 quotaPerSingularity = muldiv(
                currentPoolWeight,
                _epochTAP,
                totalWeights
            );
            // see the line below
            uint sglAssetID = sglPools[i].sglAssetID;
            // Emit weekly TAP to the pool
            // see the line below
            singularityGauges[epoch][sglAssetID] = quotaPerSingularity;
```

`_depositFeesToTwTap()` deposits to YB via `refreshPenroseFees()` and then withdraws the same amount atomically (see Penrose.sol#L511-L528):

```
function _depositFeesToTwTap(IMarket market, ITwTap twTap) private {
    if (!isMarketRegistered[address(market)]) revert NotValid();
    // see the line below
    uint256 feeShares = market.refreshPenroseFees();
    if (feeShares == 0) return;

    address _asset = market.asset();
    uint256 _assetId = market.assetId();
    yieldBox.withdraw(_assetId, address(this), address(this), 0, feeShares);

    //TODO: call twTap.distributeRewards
    uint256 rewardTokenId = twTap.rewardTokenIndex(_asset);
    uint256 feeAmount = yieldBox.toAmount(_assetId, feeShares, false);
    IERC20(_asset).approve(address(twTap), 0);
    IERC20(_asset).approve(address(twTap), feeAmount);
    twTap.distributeReward(rewardTokenId, feeAmount);
    emit LogTwTapFeesDeposit(feeShares, feeAmount);
}
```

Also BigBang.sol#L485-L503:

```
/// @notice Transfers fees to penrose
function refreshPenroseFees()
    // ...
{
    uint256 fees = asset.balanceOf(address(this));
    feeShares = yieldBox.toShare(assetId, fees, false);
    if (feeShares > 0) {
        asset.approve(address(yieldBox), fees);
        // see the line below
        yieldBox.depositAsset(
            assetId,
            address(this),
            msg.sender,
            0,
            feeShares
        );
    }
}
```

Recommendation: Consider skipping the iteration when `sglAssetID == 0`, e.g.: in TapiocaOptionBroker.sol#L714-L721:

```
  for (uint256 i; i < len; ++i) {
+     uint256 sglAssetID = sglPools[i].sglAssetID;
+     if (sglAssetID == 0) {
+         continue;
+     }
      uint256 currentPoolWeight = sglPools[i].poolWeight;
      uint256 quotaPerSingularity = muldiv(
          currentPoolWeight,
          _epochTAP,
          totalWeights
      );
-     uint sglAssetID = sglPools[i].sglAssetID;
```

Also consider adding a flag to `refreshPenroseFees()` indicating whether deposit is needed, e.g. `refreshPenroseFees(bool yieldBoxDeposit)`, and calling `refreshPenroseFees(false)` from `_depositFeesToTwTap()`.

5.5.2 `Participation` packing math is incorrect

Severity: Gas Optimization

Context: twTAP.sol#L54-L64

Description: The comments around `Participation` indicate the goal of packing the struct into 2 words

```
struct Participation {
    uint256 averageMagnitude;
    bool hasVotingPower;
    bool divergenceForce; // 0 negative, 1 positive
    bool tapReleased; // allow restaking while rewards may still accumulate
    uint56 expiry; // expiry timestamp. Big enough for over 2 billion years..
    uint88 tapAmount; // amount of TAP locked
    uint24 multiplier; // Votes = multiplier * tapAmount
    uint40 lastInactive; // One week BEFORE the staker gets a share of rewards
    uint40 lastActive; // Last week that the staker shares in rewards
}
```

However, `bool` does not occupy one byte, but rather 8. For this reason 3 slots are used:

Name	Type	Slot	Offset	Bytes	Contract
p	struct DumbContract.Participation	0	0	96	test/Test.t.sol:DumbContract
x	uint256	3	0	32	test/Test.t.sol:DumbContract

Recommendation: Consider using a bitmap for the 3 bytes.

Tapioca: Acknowledged. As far as I know, when it's stored in a struct `SSTORE` will point to the same storage slot. If it's defined outside, it will occupy an entire one.

Spearbit: Acknowledged. By using a smaller size or a bitmap you'd save one extra slot as the Struct is using 3 slots now, but it could use only 2.

5.5.3 Hardcoded `phase1Users` costs more than a merkle proof

Severity: Gas Optimization

Context: AirdropBroker.sol#L84

Description: `phase1Users` uses a mapping to store accounts and their eligible amount:

```
mapping(address => uint256) public phase1Users; /// @audit Could use merkletproof to save gas
```

This will cost 21k per account and around 5k per user claim.

Recommendation: Consider using a merkle proof like in the rest of the code.

5.5.4 `Vesting.sol` can benefit by using `immutable` variables

Severity: Gas Optimization

Context: Vesting.sol#L15-L27

Description: The following variables can be set once in the `constructor` and made immutable to save thousands of gas for end users (see Vesting.sol#L15-L27):

```
IERC20 public token; /// @audit Gas: Immutable (And add to Contructor)

/// @notice returns the cliff period
uint256 public cliff; /// @audit Gas: Immutable

/// @notice returns total vesting duration
uint256 public duration; /// @audit Gas: Immutable
```

Recommendation: Implement the refactoring by making `cliff` and `duration` immutable, and by adding `token` to the constructor and making it immutable as well.

Tapioca: Fixed in commit ece481ba7.

5.5.5 `SGLStorage` has `uint64` state variables that could be packed into a single storage slot

Severity: Gas Optimization

Context: SGLStorage.sol#L55-L58

Description: The three state variables `minimumInterestPerSecond`, `maximumInterestPerSecond`, and `startingInterestPerSecond` are all `uint64`s and are able to fit comfortably within a single 256 bit storage slot if they are declared sequentially.

Recommendation: Rearrange the declaration of these state variables so that the three `uint64`s are bundled together:

```
  uint64 public minimumInterestPerSecond;
  uint64 public maximumInterestPerSecond;
- uint256 public interestElasticity;
  uint64 public startingInterestPerSecond;
+ uint256 public interestElasticity;
```

Tapioca: Fixed in PR 322.

5.5.6 `MarketLiquidatorReceiver` is reducing allowance even in the `type(uint256).max` case

Severity: Gas Optimization

Context: MarketLiquidatorReceiver.sol#L61-L67

Description: The allowance check in `MarketLiquidatorReceiver` is as follows:

```
if (msg.sender != initiator) {
    require(
        allowances[msg.sender][tokenIn] >= collateralAmount,
        "MarketLiquidatorReceiver: sender not allowed"
    );
    allowances[msg.sender][tokenIn] -= collateralAmount;
}
```

Which is reducing the value even in the case of allowance being set to `type(uint256).max`. This will cause further gas costs and is generally not common practice.

Recommendation: Consider skipping the subtraction when the allowance is set to `type(uint256).max`.

Tapioca: Addressed in PR 322.

5.5.7 `BaseTOFTStorage` could use `immutable` values for unchangeable variables and modules

Severity: Gas Optimization

Context: BaseTOFT.sol#L114-L115 BaseTOFTStorage.sol#L25-L33

Description: The following values are not changed throughout the modules used by `TOFT`:

- BaseTOFTStorage.sol#L25-L33:

```
IYieldBoxBase public yieldBox;
/// @notice The Cluster address
ICluster public cluster;
/// @notice The ERC20 to wrap.
address public erc20;
/// @notice The host chain ID of the ERC20
uint256 public hostChainID;
/// @notice Decimal cache number of the ERC20.
uint8 internal _decimalCache;
```

Similarly, modules are set and never changed:

- BaseTOFT.sol#L114-L115

```
_leverageModule = __leverageModule;
_leverageDestinationModule = __leverageDestinationModule;
```

They could be made immutable as well.

Recommendation: Consider making them `immutable` to save massive amounts of gas.

Tapioca: Acknowledged. Did that already in V2:

```
/// @dev Used to execute certain extern calls from the TOFTv2 contract, such as ERC20Permit approvals.
TOFTv2ExtExec public immutable toftV2ExtExec;
IYieldBoxBase public immutable yieldBox;
TOFTVault public immutable vault;
uint256 public immutable hostEid;
address public immutable erc20;
ICluster public cluster;
```

Spearbit: Acknowledged.

5.5.8 Repetitive calls to `_sd2ld` can be cached to save an `SLOAD`

Severity: Gas Optimization

Context: BaseTOFTLeverageDestinationModule.sol#L42

Description: `_sd2ld` is used to convert `amountSD` to an amount in local decimals. The function will end up calling `ld2sdRate`, a storage variable. By storing the result of `_sd2ld(amountSD)` as a memory variable, an SLOAD can be saved for any subsequent read. An example of function that calls `_sd2ld` multiple times is `leverageDown`.

Recommendation: Cache the value and use `ldAmount` throughout the code:

```
uint256 ldAmount = _sd2ld(amountSD)
```

Tapioca: Fixed in PR 153.

Spearbit: Verified.

5.5.9 `Penrose.reAccrueBigBangMarkets` can be simplified to save gas

Severity: Gas Optimization

Context: Penrose.sol#L479-L480

Description: `reAccrueBigBangMarkets` checks if the market is authorized, reverts and it then checks if the caller is the `bigBangEthMarket` (see Penrose.sol#L479-L480):

```
function reAccrueBigBangMarkets() external notPaused {
    if (!isMarketRegistered[msg.sender]) revert NotAuthorized(); /// NOTE: By definition this can be
↪   skipped
    if (msg.sender == bigBangEthMarket) { /// Since the call happens only here
```

Since the accrual happens only if caller is `bigBangEthMarket`, it is possible to simply check for it, and do a no-op in any other case. This will avoid unnecessary reverts and will save gas.

Recommendation: Refactor to:

```
function reAccrueBigBangMarkets() external notPaused {
    if (msg.sender == bigBangEthMarket) {
        uint256 len = allBigBangMarkets.length;
        address[] memory markets = allBigBangMarkets;
        for (uint256 i = 0; i < len; i++) {
            address market = markets[i];
            if (market != bigBangEthMarket && isMarketRegistered[market]) {
                IBigBang(market).accrue();
            }
        }
    }
}
```

Tapioca: Updated in PR 266.

Spearbit: Verified.

5.5.10 `Penrose.hostLzChainId` is never changed - can be made `immutable`

Severity: Gas Optimization

Context: Penrose.sol#L125

Description: In Penrose.sol#L125

```
hostLzChainId = _hostLzChainId; /// @audit Gas: Immutable? Can this be changed?
```

Is never changed, and the L0 relayer also seems to use immutable values.

Recommendation: Consider making this immutable or if you have concerns for changes, consider adding a setter.

Tapioca: Made it immutable in PR 265.

Spearbit: Verified.

5.5.11 Unnecessary deposit and withdraw to and from YB

Severity: Gas Optimization

Context: Penrose.sol#L511-L528

Description: In Penrose.sol#L511-L528:

```
function _depositFeesToTwTap(IMarket market, ITwTap twTap) private {
    if (!isMarketRegistered[address(market)]) revert NotValid();

    uint256 feeShares = market.refreshPenroseFees(); /// @audit Claims fees as balanceOf
    if (feeShares == 0) return; /// But for some reason deposits it to YB

    address _asset = market.asset();
    uint256 _assetId = market.assetId();
    yieldBox.withdraw(_assetId, address(this), address(this), 0, feeShares);

    //TODO: call twTap.distributeRewards /// Then you withdraw from YB
    uint256 rewardTokenId = twTap.rewardTokenIndex(_asset);
    uint256 feeAmount = yieldBox.toAmount(_assetId, feeShares, false);
    IERC20(_asset).approve(address(twTap), 0); // So may as well just approve those tokens
    IERC20(_asset).approve(address(twTap), feeAmount);
    twTap.distributeReward(rewardTokenId, feeAmount);
    emit LogTwTapFeesDeposit(feeShares, feeAmount);
}
```

This entire flow seems to:

- Check `balanceOf`.
- Deposit it into YB.
- Re-get `balanceOf` from YB.
- Withdraw from YB.
- Send to twTAP.

Seems like the entire code could be rewritten by simply claiming the tokens and sending them to `twTap`. Also, note that directly calling `refreshPenroseFees` will cause a loss of those rewards since the function is using the return value which is based on the current `balanceOf`. Overall it seems like this can be simplified.

Recommendation: Consider refactoring to a simple transfer and call.

Tapioca: Acknowledged.

Spearbit: Acknowledged.

5.6 Informational

5.6.1 `_getInterestRate()` and `allowanceBorrow` naming is misleading, rates precision decimals are hard coded

Severity: Informational

Context: SGLCommon.sol#L37-L49, MarketERC20.sol#L79-L89, MarketERC20.sol#L96-L100, BBCollateral.sol#L38-L50, SGLCollateral.sol#L38-L50, Market.sol#L311-L312, Market.sol#L314-L315, BBCommon.sol#L31-L32, BBCommon.sol#L68-L70, BBCommon.sol#L90-L94, BigBang.sol#L146-L149, BigBang.sol#L194-L196, Singularity.sol#L187-L189, BigBang.sol#L523-L525, SGLCommon.sol#L101-L105, SGLCommon.sol#L224, SGLCommon.sol#L254

Description: _getInterestRate()‘ name looks similar to a view function returning current interest rate, while it substantially changes the state, performing a major part of interest rate and accrual logic (SGLCommon.sol#L37-L49):

```
function _getInterestRate()
    internal
    view
    returns (
        ISingularity.AccrueInfo memory _accrueInfo,
        Rebase memory _totalBorrow,
        Rebase memory _totalAsset,
        uint256 extraAmount,
        uint256 feeFraction,
        uint256 utilization,
        bool logStartingInterest
    )
{
```


`_allowedBorrow()`, `allowedBorrow()`, `allowanceBorrow` naming is somewhat misleading as in the current logic it is not a borrowing operations allowance, but rather a collateral allocation allowance (MarketERC20.sol#L96-L100):

```
/// Check if msg.sender has right to execute borrow operations
modifier allowedBorrow(address from, uint share) virtual {
    _allowedBorrow(from, share);
    _;
}
```

also MarketERC20.sol#L79-L89:

```
function _allowedBorrow(address from, uint share) internal {
    if (from != msg.sender) {
        require(
            allowanceBorrow[from][msg.sender] >= share,
            "Market: not approved"
        );
        if (allowanceBorrow[from][msg.sender] != type(uint256).max) {
            allowanceBorrow[from][msg.sender] -= share;
        }
    }
}
```

additionally BBCollateral.sol#L38-L50:

```
function removeCollateral(
    address from,
    address to,
    uint256 share
)
    external
    optionNotPaused(PauseType.RemoveCollateral)
    solvent(from, false)
    notSelf(to)
    allowedBorrow(from, share)
{
    _removeCollateral(from, to, share);
}
```

and finally SGLCollateral.sol#L38-L50:

```
function removeCollateral(
    address from,
    address to,
    uint256 share
)
    external
    optionNotPaused(PauseType.RemoveCollateral)
    solvent(from, false)
    allowedBorrow(from, share)
    notSelf(to)
{
    _removeCollateral(from, to, share);
}
```


`ratesPrecision = FEE_PRECISION_DECIMALS` of `18 - 13 = 5` is hardcoded (see Market.sol#L311-L312):

```
int256 denominator = (int256(10 ** ratesPrecision) - int256(diff)) *
    int256(1e13);
```

Also, `18` dp is used as a magic number in a number of cases (see Market.sol#L314-L315):

```
//compute closing factor
int256 x = (int256(numerator) * int256(1e18)) / denominator;
```

- BBCommon.sol#L31-L32:

  ```
  uint256 _maxDebtPoint = (_ethMarketTotalDebt *
      debtRateAgainstEthMarket) / 1e18;
  ```

- BBCommon.sol#L68-L70:

  ```
  uint256 extraAmount = (uint256(_totalBorrow.elastic) *
      uint64(getDebtRate() / 31536000) *
      elapsedTime) / 1e18;
  ```

- BBCommon.sol#L90-L94:

  ```
  extraAmount =
      (uint256(_totalBorrow.elastic) *
          _accrueInfo.debtRate *
          elapsedTime) /
      1e18;
  ```

- BigBang.sol#L146-L149:

```
if (minDebtRate != 0 && maxDebtRate != 0) {
    if (_debtRateMin >= _debtRateMax) revert DebtRatesNotValid();
    if (_debtRateMax > 1e18) revert MaxDebtRateNotValid();
}
```

- BigBang.sol#L194-L196:

```
EXCHANGE_RATE_PRECISION = _exchangeRatePrecision > 0
    ? _exchangeRatePrecision
    : 1e18;
```

- Singularity.sol#L187-L189:

```
EXCHANGE_RATE_PRECISION = _exchangeRatePrecision > 0
    ? _exchangeRatePrecision
    : 1e18;
```

- BigBang.sol#L523-L525:

```
if (_maxDebtRate > 0) {
    if (_maxDebtRate <= minDebtRate) revert DebtRatesNotValid();
    if (_maxDebtRate > 1e18) revert DebtRatesNotValid();
```

- SGLCommon.sol#L101-L105:

```
extraAmount =
    (uint256(_totalBorrow.elastic) *
        _accrueInfo.interestPerSecond *
        elapsedTime) /
    1e18;
```

- SGLCommon.sol#L224-L226:

```
if (_totalAsset.base + uint128(fraction) < 1000) {
    return 0;
}
```

- SGLCommon.sol#L253-L254:

```
_totalAsset.base -= uint128(fraction);
if (_totalAsset.base < 1000) revert MinLimit();
```

Recommendation: Consider:

- Renaming the `_getInterestRate()`, e.g. to `_getAccruedAndUpdatedState()`.
- Renaming these variables uniformly across the codebase, e.g. using `_allowedCollateral()`, `allowedCollateral()`, `allowanceCollateral`.
- Replacing the decimals related magic numbers with the corresponding constants (see Market.sol#L99-L100):

```
uint256 internal constant FEE_PRECISION = 1e5;
uint256 internal constant FEE_PRECISION_DECIMALS = 5;
```

E.g. consider defining `18` as `BASE_PRECISION_DECIMALS` and use `10**(BASE_PRECISION_DECIMALS - FEE_PRECISION_DECIMALS)` instead of `1e13`. Similarly, consider replacing `1e18` with a constant, e.g. using `DEBT_PRECISION` (see BBStorage.sol#L55):

```
uint256 internal constant DEBT_PRECISION = 1e18;
```

For the minimum amount limits, consider replacing `1000` magic number with the corresponding constant, taking asset decimals into account. Apart from decimals different assets might have different valuations and it is recommended making the definition of dust / minimum adjustable.

5.6.2 Incorrect error handling will miss custom errors with 0 or 1 parameters

Severity: Informational

Context: Penrose.sol#L496, Market.sol#L402, BaseUSDOStorage.sol#L85, BaseTOFTStorage.sol#L106

Description: `_getRevertMsg` is used throughout the codebase and will work when catching any `string` errors. As a string returned by an external call will always have at least 2 words when in memory, and 3 words when in calldata plus the error selector (for a total of 68 in length).

However, a custom error is comprised of 4 bytes and one word when it has 0 or 1 parameters, meaning that the function will not catch custom errors of that length.

- **Proof of concept:**

```
// SPDX-License Identifier: MIT

pragma solidity 0.8.17;

import "forge-std/Test.sol";
import "forge-std/console2.sol";


library ErrorParser {
    function _getRevertMsg(
        bytes memory _returnData
    ) internal pure returns (string memory) {
        // If the _res length is less than 68, then the transaction failed silently (without a
↪  revert message)
        if (_returnData.length < 68) return "USDO: data";
        // solhint-disable-next-line no-inline-assembly
        assembly {
            // Slice the sighash.
            _returnData := add(_returnData, 0x04)
        }
        return abi.decode(_returnData, (string)); // All that remains is the revert string
    }
}
contract CustomErrorOutLongLength {

    error ARandomErrror(uint256 x, uint256 y);

    function doTheCall() external {
        revert ARandomErrror(1, 1);
    }
}
contract CustomErrorNormal{

    error ARandomErrror();

    function doTheCall() external {
        revert ARandomErrror();
    }
}

contract ErrorCallerContract {
    // This extra processing will mess up the one below
    function doTheCall(ErrorOut target) external {
```

```
            try target.doTheCall() {} catch (bytes memory reason) {
                revert(ErrorParser._getRevertMsg(reason));
            }

        }
    }

    contract ExampleTest is Test {


        // 0 or 1 param will pass as the error is not parsed
        function testCustomErrorNormal() public {
            CustomErrorNormal reverter = new CustomErrorNormal();

            try reverter.doTheCall() {} catch (bytes memory reason) {
                string memory errorMessage = ErrorParser._getRevertMsg(reason);
                assertEq(keccak256("USDO: data"), keccak256(abi.encodePacked(errorMessage)), "Same
↪   hash");
            }
        }

        // 2 or more param will fail as the error is correctly handled
        function testCustomErrorLong() public {
            CustomErrorOutLongLength reverter = new CustomErrorOutLongLength();

            try reverter.doTheCall() {} catch (bytes memory reason) {
                string memory errorMessage = ErrorParser._getRevertMsg(reason);
                assertEq(keccak256("USDO: data"), keccak256(abi.encodePacked(errorMessage)), "Same
↪   hash");
            }
        }

    }
```

Recommendation: Consider changing the length check to work with custom errors or whether you want to only support error strings.

5.6.3 `AllowanceNotValid` custom error used for insufficient emissions

Severity: Informational

Context: TapOFT.sol#L259-L260

Description: The custom error `AllowanceNotValid` is used throughout the codebase to imply allowance issue. However, it is used in `extractTAP` to imply that emissions are insufficient:

```
if (emissionForWeek[week] < mintedInWeek[week] + _amount)
    revert AllowanceNotValid(); /// @audit QA: Error looks off, should be InsufficientEmissions
```

Recommendation: Consider using a different custom error.

Tapioca: Defined a new error message:

```
if (emissionForWeek[week] < mintedInWeek[week] + _amount) {
    revert InsufficientEmissions();
}
```

5.6.4 `twAML` related invariant testing recommendation

Severity: Informational

Context: twTAP.sol

Description/Recommendation: `twTAP` is a sophisticated component of Tapioca's Governance. It's based on allowing the free market to extend and contract locking durations as means to unlock higher and lower locks.

Due to the relative nature of said weights, as well as the clear spec, it's highly recommended that the core logic of `twTAP` is invariant tested as to ensure the following properties:

- No lock can be created and unlocked in the same block.
- No lock can be created and unlocked before its (original, `uint256`) duration has expired.
- Given a small amount of dust, I can never reach X multiple.
- I can never lock for longer than X (intended soft max cap enforced by math if not added via `require`).
- The sum of all balances at a given epoch, is the sum of all active locks.
- If I could have unlocked at epoch X, then the `totalSum` of Balances at epoch X reflects that.

We would recommend refactoring the logic of `TWAML`, `twTAP` and `TapiocaOptionsBroker` to be re-used. The logic for magnitude, cumulative etc, is very similar and can be re-used too. This will allow invariant testing on the core logic without much overhead.

5.6.5 `AirdropBroker` nitpicks

Severity: Informational

Context: AirdropBroker.sol

Description: The following small tweaks could be applied to improve code quality:

1. In AirdropBroker.sol#L63-L64, move `tapOracle` and `oracleData` closer:

```
IOracle public tapOracle; /// @audit QA/R: move `oracle` and `oracleData` closer for logical
↪  adjacency
```

2. In AirdropBroker.sol#L94-L95, refactor discounts to all be in BPS:

```
uint8[4] public PHASE_2_DISCOUNT_PER_USER = [50, 40, 40, 33]; /// @audit Make it consistent
```

3. In AirdropBroker.sol#L331, consider calling `aoTAP.brokerClaim()` in the constructor

```
function aoTAPBrokerClaim() external { /// @audit May want to call this on deployment
```

Recommendation: Consider applying the aforementioned refactorings.

Tapioca: Regarding the refactorings:

1 - Doesn't exist anymore on V2 migration. 2 - Defined `uint24[4] public PHASE_2_DISCOUNT_PER_USER = [500_000, 400_000, 330_000, 250_000]`; 3 - Yes, it is

5.6.6 Inconsistent logic in setting `activeSingularities[singularity].poolWeight`

Severity: Informational

Context: TapiocaOptionLiquidityProvision.sol#L285-L286

Description: `setSGLPoolWEight` behaves differently from `registerSingularity`: `setSGLPoolWEight` allows setting any value.

- TapiocaOptionLiquidityProvision.sol#L285-L286:

```
activeSingularities[singularity].poolWeight = weight; /// @audit QA: Not consistent
```

While `registerSingularity` (TapiocaOptionLiquidityProvision.sol#L320-L321):

```
activeSingularities[singularity].poolWeight = weight > 0 ? weight : 1; /// @audit QA: Not consistent
```

Enforces a minimum weight of 1.

Recommendation: Add a comment on the asymmetry.

Tapioca: Acknowledged.

Spearbit: Acknowledged.

5.6.7 `_getDiscountedPaymentAmount` has an incorrect comment regarding precision

Severity: Informational

Context: TapiocaOptionBroker.sol#L677-L678

Description: `_getDiscountedPaymentAmount` asserts that it uses discount in BPS.

- TapiocaOptionBroker.sol#L677-L678

```
/// @param _discount The discount in BPS
```

However, it will divide the divide it by `100e4`

- TapiocaOptionBroker.sol#L689-L690

```
muldiv(_otcAmountInUSD, _discount, 100e4); // 1e4 is discount decimals, 100 is discount
↪ percentage
```

Meaning that discount is hundreds of basis points.

Recommendation: Update the comment or the unit used.

5.6.8 The keyword `average` is used improperly in `pool.averageMagnitude`

Severity: Informational

Context: twTAP.sol#L317-L319

Description: The code uses the keyword average, while it's not computing an average:

```
pool.averageMagnitude =
    (pool.averageMagnitude + magnitude) /
    pool.totalParticipants; // compute new average magnitude | // new Magnitude / total? /// @audit
↪ This is NOT average
```

Recommendation: Consider rephrasing the line, or adding further documentation around the variable choices. E.g. `magnitudeStep` can be a better name as the variable essentially represents the current adjustment step the system makes with a new entry.

5.6.9 Reported issues across reviewed codebases and upcoming changes may lead to unexpected behavior

Severity: Informational

Context: Global scope

Description: The protocol has had four security reviews so far:

1. Certora (April 2023): The scope of this review was the formal verification of `YieldBox.sol` and reported 3 High + 2 Medium issues.
2. Code4rena (September 2023): The scope of this review included everything from our Spearbit review along with `tapioca-periph` and `tapioca-yieldbox-strategies`, and reported 60 High + 99 Medium issues.
3. Pashov Audit Group (January 2024): The scope of this review included Stargate, Magnetar, Swapper and oracle components, and reported 3 Critical + 7 High + 16 Medium issues.
4. Spearbit Review (January 2024): The scope of our review was similar to the Code4rena review excluding the `tapioca-periph` and `tapioca-yieldbox-strategies` (and also excluding Leverage executors and `BaseTOFTStrategy` modules) to serve as a fix review for Code4rena issues and the refactoring thereafter, and reporting 7 Critical + 31 High + 57 Medium issues.

From the current implementation, the protocol team, among other things, plans to:

1. Fix the issues reported in Pashov Audit Group and this Spearbit review.
2. Upgrade the entire protocol from the current LayerZero V1 to the recently announced LayerZero V2, which should significantly affect interactions with the base layer for cross-chain operations.
3. Revisit some design aspects related to cross-chain crypto-economic concerns raised during this review's discussions.
4. Port over their development infrastructure from Hardhat to Foundry.

Impact: The final codebase may have latent issues due to:

1. Time-bounded best-effort security reviews across large codebases with significant complexity.
2. Lack of a single comprehensive review of the entire protocol leading to compositional issues across reviewed codebases.
3. Number and nature of systemic issues reported across reviews.
4. Extent of fixes resulting in regressions from and interactions between fixes to reported issues.
5. Upgrade to LayerZero V2.
6. Potential redesigns to cross-chain operations and YieldBox integration.
7. Insufficient specification and detailed documentation for all implemented user flows.
8. Insufficient testing as reported in the issue "Insufficient testing may lead to unexpected behavior".

Recommendation:

1. Provide sufficient specification and detailed documentation for all implemented user flows.
2. Implement validation recommendations as suggested in the issue "Insufficient testing may lead to unexpected behavior".
3. Pay special attention to cross-chain operations and its dependencies on LayerZero and assumptions on UI/UX (e.g. issue "Users with a smart-contract wallet address may have funds drained from an attacker who controls that address on another chain").
4. Consider one or more comprehensive reviews of the entire protocol after all upgrades and fixes are in place.
5. Adopt a guarded launch approach with respect to deployed tokens, markets, chains and TVL.
6. Ensure a bug bounty is in place beforehand.

7. Consider monitoring solutions.
8. Ensure an incident response plan is in place.

Tapioca: Acknowledged.

Spearbit: Acknowledged.

5.6.10 Insufficient testing may lead to unexpected behavior

Severity: Informational

Context: Global scope

Description: Testing against expected behavior as documented in a specification is critical. This includes unit testing, integration testing, fuzz testing, property-based invariant testing, regression testing and E2E testing among other forms of validation. Using tools for static analysis and fuzzing integrated into the development lifecycle to test for well-known vulnerabilities or other unexpected behavior is also necessary. This is especially required for projects such as this one with several components, complex logic, multiple tokens, cross-chain and several layers of interactions both within and across chains resulting in a large attack surface.

However, it is not evident that sufficient testing has been performed given the number and nature of issues raised in this and other reviews.

Impact: Insufficient testing may lead to unexpected behavior.

Recommendation: Ensure unit testing, integration testing, fuzz testing, property-based invariant testing, regression testing and E2E testing among other forms of validation are built into the CI/CD pipeline of the project. Some specific suggestions are made in the issues "Invariant testing recommendations" and "E2E LayerZero testing checklist".

Tapioca: Acknowledged.

Spearbit: Acknowledged.

5.6.11 Unusual `setTokenURI` usage

Severity: Informational

Context: aoTAP.sol#L103-L106, oTAP.sol#L104-L111

Description: The `aoTap` and `oTap` contracts present an unusual pattern for `setTokenURI` which allows the owner to set the URI to any value (see oTAP.sol#L105-L111):

```
function setTokenURI(uint256 _tokenId, string calldata _tokenURI) external {
    require(
        _isApprovedOrOwner(msg.sender, _tokenId),
        "OTAP: only approved or owner"
    );
    tokenURIs[_tokenId] = _tokenURI;
}
```

This will not cause any specific vulnerability, however, it's worth keeping in mind that marketplaces, such as Opensea, will render something out of the `tokenURI`, which may be used to deceive people. Additionally, the arbitrary data will allow for values to be wildly different between `tokenId`s.

Recommendation: Hardcode the `tokenURI`, or consider using a `json` or `svg` rendered.

Tapioca: Acknowledged. It was an early concept. We haven't come with a final concept on this one.

Spearbit: Acknowledged.

5.6.12 `Vesting.sol` QA findings

Severity: Informational

Context: Vesting.sol#L41-L42

Description: In reviewing `Vesting.sol` the following informational findings were found

1. `__totalAmount` cannot be fetched via etherscan, consider making it public (see Vesting.sol#L41-L42):

```
uint256 private __totalAmount; /// @audit QA: No way to fetch this on Etherscan
```

2. `_initialUnlock` is meant to be capped to `10_000` (100%), add a check to prevent incorrect settings (see Vesting.sol#L209-L210):

```
uint256 _initialUnlock /// @audit QA: Cap to 10_000
```

Recommendation: Implement the refactorings.

Tapioca: Fixed in commit b16f25c9a

Spearbit: Verified.

5.6.13 Function/variable names not accurately reflecting their actions/state affects readability

Severity: Informational

Context: Penrose.sol#L530-L565, Penrose.sol#L377-L393, Penrose.sol#L398-L407, BaseLeverageExecutor.sol#L117, SGLCommon.sol#L37

Description: Function/variable names should accurately reflect their actions/state so that developers, users and reviewers do not miss/misunderstand their semantics and side-effects.

There are several functions/variables across the codebase whose names do not accurately indicate their entire/correct functionality. For example:

1. `_getMasterContractLength()` is neither a private/internal function (which the leading _ is supposed to indicate per Solidity style guidelines) nor does it return the master contract length. It is a public function which returns all the markets for all the master contracts in the array argument.
2. `Penrose.registerSingularity()` could be `Penrose.deployAndRegisterSingularity()` because it also deploys besides registering.
3. `Penrose.addSingularity()` could be `Penrose.registerSingularity()` because it only registers.
4. `gas` is actually `value`, recommend renaming.
5. Consider refactoring the name to something like `_getAccruedAndUpdatedState()` as now the naming is somewhat misleading (i.e. it looks similar to a view function returning current interest rate).

Impact: Function/variable names not accurately reflecting their actions/state affects readability.

Recommendation: Consider using function/variable names that accurately reflect their actions/state. If the names become too long/confusing for that reason, consider an alternative naming scheme that is consistently applied across the codebase.

Tapioca: Acknowledged.

Spearbit: Acknowledged.

5.6.14 Unused code constructs indicate missing/stale functionality and affect readability

Severity: Informational

Context: Penrose.sol#L41, Penrose.sol#L156-L161, Market.sol#L76, SGLStorage.sol#L121, SGLStorage.sol#L156, SGLStorage.sol#L161, BBCommon.sol#L139, SGLCommon.sol#L278, SGLLendingCommon.sol#L36, SGLLiquidation.sol#L158, BaseTapOFT.sol#L455-L477

Description: There are several unused variables, events, custom errors and functions across the codebase. These either indicate missing functionality which is yet to be implemented or stake functionality which can be removed.

Impact: Presence of unused code constructs indicates missing/stale functionality and affects readability.

Recommendation: Consider implementing the missing functionality or removing these stale code constructs.

Tapioca: Event was removed in a previous PR.

5.6.15 Vestigial `_PERMIT_TYPEHASH_DEPRECATED_SLOT` for non-upgradeable contracts

Severity: Informational

Context: MarketERC20.sol#L36-L43

Description: `MarketERC20` are non upgradeable contracts. The deployment of Tapioca is going to be a fresh one. However, `MarketERC20` contains this variable, with the following comment:

```
/**
 * @dev In previous versions `_PERMIT_TYPEHASH` was declared as `immutable`.
 * However, to ensure consistency with the upgradeable transpiler, we will continue
 * to reserve a slot.
 * @custom:oz-renamed-from _PERMIT_TYPEHASH
 */
// solhint-disable-next-line var-name-mixedcase
bytes32 private _PERMIT_TYPEHASH_DEPRECATED_SLOT;
```

Which implies having to maintain upgradeability.

Recommendation: Consider removing the code and ensure tests are capturing changes in contract slot.

Tapioca: Fixed in PR 322.

Spearbit: Verified.

5.6.16 `TapiocaWrapper` allows deploying only one between `tOFT` and `mOFT` per underlying

Severity: Informational

Context: TapiocaWrapper.sol#L141-L143

Description: `TapiocaWrapper.createTOFT` has the following check (see TapiocaWrapper.sol#L141-L143):

```
if (address(tapiocaOFTsByErc20[_erc20]) != address(0x0)) {
    revert TapiocaWrapper__AlreadyDeployed(_erc20);
}
```

Which will prevent deploying any additional `tOFT` or `mtOFT` for a given ERC20. This will prevent deploying an `mtOFT` for a token that has a `tOFT` without deploying a new `TapiocaWrapper`.

Recommendation: Consider whether you should allow both types of contracts to be deployed.

Tapioca: Acknowledged. We will have only 1 contract type for an underlying asset.

Spearbit: Acknowledged.

5.6.17 `BaseTOFT._nonblockingLzReceive` `delegatecall`s to `address(0)` when module is not found

Severity: Informational

Context: BaseTOFT.sol#L638-L640

Description: `_nonblockingLzReceive` has a "switch-like" logic that will result in `targetModule = address(0);` in the default case (see BaseTOFT.sol#L638-L640):

```
} else {
    targetModule = address(0);
}
```

It will then `delegatecall` to that address with all the parameters. Since the code is meant to cause a no-op, it may be best to revert instead.

Recommendation: Revert when a module is not found.

Tapioca: Acknowledged. Did it with V2:

```
} else {
    revert InvalidMsgType(msgType_);
}
```

Spearbit: Acknowledged.

5.6.18 `BaseTOFT` is not charging any fee despite the comment

Severity: Informational

Context: TapiocaOFT.sol#L60-L75, BaseTOFT.sol#L546-L559

Description: The comment on `wrap` for both `TapiocaOFT` and `mTapiocaOFT` is as follows (see TapiocaOFT.sol#L60-L75):

```
/// @notice Wrap an ERC20 with a 1:1 ratio with a fee if existing.
/// @dev Since it can be executed only on the main chain, if an address exists on the OP chain it will
↪  not allowed to wrap.
/// @param _fromAddress The address to wrap from.
/// @param _toAddress The address to wrap the ERC20 to.
/// @param _amount The amount of ERC20 to wrap.
function wrap(
    address _fromAddress,
    address _toAddress,
    uint256 _amount
) external payable onlyHostChain {
    if (erc20 == address(0)) {
        _wrapNative(_toAddress);
    } else {
        _wrap(_fromAddress, _toAddress, _amount);
    }
}
```

However, no fee is charged as of now (see BaseTOFT.sol#L546-L559):

```
function _wrap(
    address _fromAddress,
    address _toAddress,
    uint256 _amount
) internal virtual {
    if (_fromAddress != msg.sender) {
        if (allowance(_fromAddress, msg.sender) < _amount)
            revert AllowanceNotValid();
        _spendAllowance(_fromAddress, msg.sender, _amount);
    }
    if (_amount == 0) revert NotValid();
    IERC20(erc20).safeTransferFrom(_fromAddress, address(vault), _amount);
    _mint(_toAddress, _amount);
}
```

Recommendation: Define mechanisms for charging a fee and apply them to the contracts.

Tapioca: Acknowledged. Fixed the natspec in v2. Only mTOFT charges a fee.

Spearbit: Acknowledged.

5.6.19 Misplaced input validation deviates from CEI and affects readability

Severity: Informational

Context: AirdropBroker.sol#L553

Description: There are many places in the protocol where input validation of parameters is not performed at the beginning of their functions but is mixed up with other function logic. This not only deviates from the recommended checks-effects-interactions (CEI) guidelines but also affects readability.

Recommendation: Consider moving all the input validation of function parameters to the beginning of functions.

Tapioca: Addressed in commit dba79d4ca.

5.6.20 Checking `uint` variables against `<= 0` affects readability

Severity: Informational

Context: AirdropBroker.sol#L500, TapiocaOptionBroker.sol#L474, TapiocaOptionBroker.sol#L623, TapiocaOptionBroker.sol#L686

Description: Given that `uint` variables can never be less than zero, there is no need to check them against `<= 0`. Doing so affects readability.

Recommendation: Consider changing such occurrences to `== 0` check instead.

Tapioca: Addressed in commit 29f3c3fd7.

5.6.21 Second phase of `aoTAP` distribution mentions a different Tapioca Guild Role in code comment versus documentation

Severity: Informational

Context: AirdropBroker.sol#L91, AirdropBroker.sol#L497, Tapioca Documentation

Description: The second phase of `aOTAP` distribution mentions "Cassava" as the fourth Tapioca Guild Role in code comment but the documentation has this as "Sushi Frens." Both however mention "Cassava" as part of the fourth phase.

Impact: It is unclear if "Cassava" is also the intended recipient in second phase.

Recommendation: Correct the code comment or documentation to keep them in sync.

5.6.22 `strategyDeposit` incorrect variable name

Severity: Informational

Context: BaseTOFTStrategyDestinationModule.sol#L49-L52

Description: `strategyDeposit` is meant to set tokens to `to` which is encoded in `BaseTOFTStrategyModule.sendToStrategy` as follows (see BaseTOFTStrategyModule.sol#L71-L79):

```
bytes memory lzPayload = abi.encode(
    PT_YB_SEND_STRAT,
    LzLib.addressToBytes32(_from),
    toAddress,
    _ld2sd(amount),
    assetId,
    options.zroPaymentAddress
);
```

When abi.decoding, the first parameter will be `PT_YB_SEND_STRAT`, the second will be `_from` and the third will be `toAddress`. However in `strategyDeposit` the 3rd parameter is labeled as `from` (see BaseTOFTStrategyDestinationModule.sol#L49-L52):

```
(, , bytes32 from, uint64 amountSD, uint256 assetId, ) = abi.decode(
    _payload,
    (uint16, bytes32, bytes32, uint64, uint256, address)
);
```

Recommendation: Rename `from` → `to`.

Tapioca: Module was removed in a subsequent PR.

Spearbit: Verified.

5.6.23 E2E LayerZero testing checklist

Severity: Informational

Context: Global scope

Description: The following is an incomplete checklist of tests that should be performed against the deployed and setup system.

Checklist:

- Does the Call Work with the correct parameters?
- Does the Call get caught properly on failure cases?
- Does the Call work on retry?
- Are `minDstGasLookup` setup, for each dst and src chain to prevent reverts?

- Can any call inject arbitrary addresses?
- Can we reenter on any call?
- Can we pass malformed data?
- Can we burn more gas than intended and make the nonBlockingApp revert?
  - By return bombing
  - By revert bombing
  - By burning all gas / gas griefing
- Can we make an uncaught call revert?
  - By self destructing
  - By return bombing
  - By revert bombing
  - By burning all gas / gas griefing
- Can we perform operations on other people behalf? (even if they result in net-zero changes)
- Can we delay or deny other people operations?
- Can our actions, in any way put others at risk?

Cross Chain Ownership: Per the Wintermute Exploit, we know that xChain ownerhip of a specific address cannot be guaranteed.

- Can a xChain call, allow owner A to steal tokens from owner B by "mining" owner Bs address?

Funsig Delegation Setup: All modules should be tested to ensure that they work and behave as intended. A full E2E set of tests could be setup via synpress.

Recommendation: Consider extending this checklist and verifying each point in a way that is both:

- Thorough.
- As automated as possible.

As to ensure high coverage while avoiding this verification becoming too burdensome. Ensure you will perform a security review once all contracts, and L0 setup has been completed as to ensure that the final settings are safe.

5.6.24 Invariant testing recommendations

Severity: Informational

Context: Tapioca-bar

Description: The following is an incomplete list of recommended invariant checks to be setup through invariant testing.

Miro Board.

Initial Resources:

- Intro to Fuzzing.
- Advanced Fuzzing
- Trail of Bits full tutorial
- Tips to Master Fuzzing

Comments:

- For any Public / External Function, write a handler.
- Consider if it's necessary to also add handlers for admin functions (probably clamped values).
- Must add donations of:
  - Tokens → Yieldbox `token.transfer(yieldbox, amt)`.
  - Yieldbox (via shares and via amount) → SGL / BB `yb.transfer(bb, amt)` | `yb.transferShares(bb, amt)`.
- Consider custom oracle with extra revert clauses BraindeadFeedUnit.t.sol#L9-L163.
- Price changes should be pretty random as to test stale and unstale price feeds

Users - Track these:

- Deposited shares and amounts.
  - Before and after check that they match (per invariant on Yieldbox).
- Deposited Collateral to SGL.
- Deposited Asset to SGL.
- Borrowed Asset to SGL.
- If user is liquidated, you will have to adapt to check the new values.
  - Specifically to ensure how some premium is paid and if any tokens will be sent back to the user.

Global - Track these:

- LTV / HF of system.
  - Must improve when liquidations happen.
  - Must never go below X if there are no liquidations available.
- Liquidations.
  - Before/after of system.
  - Before/after of liquidator.
  - Update user data to track properly.

Maybe best to have a separate account do liquidations as to track them separately, I believe one address can do all liquidations, no need for more

Examples (pseudocode):

- `addCollateral` (BigBang.sol#L244-L262):

```
  function addCollateral(
      address from,
      address to,
      bool skim,
      uint256 amount,
      uint256 share
  ) external {
      _executeModule(
          Module.Collateral,
          abi.encodeWithSelector(
              BBCollateral.addCollateral.selector,
              from,
              to,
              skim,
              amount,
              share
          )
      );
  }
```

```
  function addCollateralToSelfAsAmount(uint256 amount) external {
    uint256 balanceBefore = yb.balanceOf(address(this));
    sgl.addCollateral(address(this), address(this), false, amount, 0);
    uint256 balanceAfter = yb.balanceOf(address(this));
    uint256 deltaBalance = balanceBefore - balanceAfter;
    userDeposits += deltaBalance;
    // TODO: Also compare the change in sgl storage balance value to ensure it's consistent with
↪   what was paid
  }

  function addCollateralToSelfAsShares(uint256 shares) external {
    uint256 balanceBefore = yb.balanceOf(address(this));
    sgl.addCollateral(address(this), address(this), false, 0, shares);
    uint256 balanceAfter = yb.balanceOf(address(this));
    userDeposits += balanceBefore - balanceAfter;
  }


  // To compare the two, you'd have to write a contract that reverts
  // See: https://github.com/ebtc-protocol/ebtc/blob/feat/release-0.5/packages/contracts/contracts
↪   /CRLens.sol
```


- `repay` (BigBang.sol#L308-L325):

```
function repay(
    address from,
    address to,
    bool skim,
    uint256 part
) external returns (uint256 amount) {
    bytes memory result = _executeModule(
        Module.Borrow,
        abi.encodeWithSelector(
            BBBorrow.repay.selector,
            from,
            to,
            skim,
            part
        )
    );
    amount = abi.decode(result, (uint256));
}
```

```
function repaySelf(uint256 part) external {
  uint256 amtRepaid = sgl.repay(address(this), address(this), false, part);
  // TODO: Verify that repaid amt is == owed (including interest)
  // Or that is the amt borrowed if interest is 0 (good starting point initially)
}
```

BB Invariants:

- **Critical - Value of basket changes properly**

  Given deposits and withdrawals the total value of basket changes accordingly to the value that is being moved (both for shares and amounts)

  At all times (DOUBLE CHECK):

  - `toAmount(shares, false)` → could be withdrawn from the system.
  - `toShares(amount, true)` → is in the system.
- **Critical - Rounding of values don't cause basket value to not change**
- **Critical - Value of basket is real value of basket** (After withdrawing 100%, we get the same amount)
- **Global - Critical - Relative value of Basket only changes via interest**
  - Value of basket must not change beside deflating due to interest, else it means value can be stolen.
- Global - Leverage /Macros/etc.. Must always respect the same global invariants.
- BBBorrow:
  - **Low - Opening Fee math is reasonable**: BBBorrow.sol#L39-L40

```
uint256 feeAmount = _computeVariableOpeningFee(amount);
```

  - Opening Fee is a % of amount.
  - Opening Fee is paid to XYZ.
  - **Low - Allowance math is correct:** BBBorrow.sol#L40-L45

```
    uint256 allowanceShare = _computeAllowanceAmountInAsset(
        from,
        exchangeRate,
        amount + feeAmount,
        asset.safeDecimals()
    );
```

```
  - Alowance math is the exact amount that ends up being used
```

  - **Low - Allowance math 2:** BBBorrow.sol#L47-L48

```
_allowedBorrow(from, allowanceShare);
```

```
  - Given X Allowance
  - I can move X Asset
```

  - **High - Math on Borrow**: BBBorrow.sol#L76-L77

```
(_totalBorrow, partInAmount) = _totalBorrow.sub(part, true);
```

  - **High - Self Liquidation is prevented**
  - **High - Self Liquidation via accrual as well**

- BBCollateral:
  - **Medium - Math on `amount` and `share` - M-AS:** BBCollateral.sol#L26-L28

```
if (share == 0) {
    share = yieldBox.toShare(collateralId, amount, false);
}
```

  - Given an amount, I can only receive a specific quantity of shares.
  - Given shares, I can only receive a specific quantity of amount.

  How: - Deposit in YB → Deposit Shares into BB. vs - Deposit in YB → Deposit Amount into BB.

BBCommon

- **Medium - DebtRate is within bounds:** BBCommon.sol#L23-L45

```
function getDebtRate() public view returns (uint256) {
    if (isMainMarket) return penrose.bigBangEthDebtRate(); // default 0.5%
    if (totalBorrow.elastic == 0) return minDebtRate;

    uint256 _ethMarketTotalDebt = IBigBang(penrose.bigBangEthMarket())
        .getTotalDebt();
    uint256 _currentDebt = totalBorrow.elastic;
    uint256 _maxDebtPoint = (_ethMarketTotalDebt *
        debtRateAgainstEthMarket) / 1e18;

    if (_currentDebt >= _maxDebtPoint) return maxDebtRate;

    uint256 debtPercentage = ((_currentDebt - debtStartPoint) *
        DEBT_PRECISION) / (_maxDebtPoint - debtStartPoint);
    uint256 debt = ((maxDebtRate - minDebtRate) * debtPercentage) /
        DEBT_PRECISION +
        minDebtRate;

    if (debt > maxDebtRate) return maxDebtRate;

    return debt;
}
```

- **High - getDebtRate never reverts under any condition, for any reason**
  - Have a caller that `try catches` to `getDebtRate` and assert `success`
- **Medium - Should never revert:** BBCommon.sol#L55-L72


```
function _accrueView()
    internal
    view
    override
    returns (Rebase memory _totalBorrow)
{
    uint256 elapsedTime = block.timestamp - accrueInfo.lastAccrued;
    if (elapsedTime == 0) {
        return totalBorrow;
    }

    // Calculate fees
    _totalBorrow = totalBorrow;
    uint256 extraAmount = (uint256(_totalBorrow.elastic) *
        uint64(getDebtRate() / 31536000) *
        elapsedTime) / 1e18;
    _totalBorrow.elastic += uint128(extraAmount);
}
```

- **Medium - _accrueView should result in the same value as** `accrue`
  - Call `_accrueView`, get the values.
  - Call `accrue`, compare the values.
- **High - Accrue should never revert:** BBCommon.sol#L74-L101

```
function _accrue() internal override {
    IBigBang.AccrueInfo memory _accrueInfo = accrueInfo;
    // Number of seconds since accrue was called
    uint256 elapsedTime = block.timestamp - _accrueInfo.lastAccrued;
    if (elapsedTime == 0) {
        return;
    }
    //update debt rate
    uint256 annumDebtRate = getDebtRate();
    _accrueInfo.debtRate = uint64(annumDebtRate / 31536000); //per second
    _accrueInfo.lastAccrued = uint64(block.timestamp);

    Rebase memory _totalBorrow = totalBorrow;

    // Calculate fees
    uint256 extraAmount = 0;
    extraAmount =
        (uint256(_totalBorrow.elastic) *
            _accrueInfo.debtRate *
            elapsedTime) /
        1e18;
    _totalBorrow.elastic += uint128(extraAmount);

    totalBorrow = _totalBorrow;
    accrueInfo = _accrueInfo;

    emit LogAccrue(extraAmount, _accrueInfo.debtRate);
}
```

- **High - Accrue Math should be sound**
  - Should never increase the amount above time * rate (SPEC)
  - Should be idempotent (call it twice / three time), no change in result
- **Medium - Accrue math should never overflow**
  - Should never overflow (overflow by checking that `extraAmount` doesn't overflow).
  - Add a casting and check for revert.
  - Add a specific check for overflow and assert and then revert (basically same thing).
- **Low - Add Tokens can never grant more results than what was added:** BBCommon.sol#L111-L127

```
function _addTokens(
    address from,
    uint256 _tokenId,
    uint256 share,
    uint256 total,
    bool skim
) internal {
    if (skim) {
        require(
            share <= yieldBox.balanceOf(address(this), _tokenId) - total,
            "BB: too much"
        );
    } else {
        yieldBox.transfer(from, address(this), _tokenId, share);
    }
}
```

Check share, do donation, verify this never goes above. This handler could be a pain to code as a donation may happen before

BBLendingCommon:

- **Medium - M-AS: BBLendingCommon.sol#L28-L30

```
if (share == 0) {
    share = yieldBox.toShare(collateralId, amount, false);
}
```

- **Medium - Shares changes are consistent:** BBLendingCommon.sol#L34-L35

```
_addTokens(from, collateralId, share, oldTotalCollateralShare, skim);
```

  - Compare a call to `share = yieldBox.toShare(collateralId, amount, false);`
  - Verify that `yieldBox.balanceOf(this)` changes accordingly.

- **Medium - Share balance changes consistently:** BBLendingCommon.sol#L31

```
userCollateralShare[to] += share;
```

  Verify similar to above.

- 3 Mediums - Same as above: BBLendingCommon.sol#L39-L48

```
function _removeCollateral(
    address from,
    address to,
    uint256 share
) internal {
    userCollateralShare[from] -= share;
    totalCollateralShare -= share;
    emit LogRemoveCollateral(from, to, share);
    yieldBox.transfer(address(this), to, collateralId, share);
}
```


- **Low - Fee Math is consistent**:
  - Whatever the fee preview is, is the change to `to`.
- **Low - Accounting of** `userBorrowPart`: BBLendingCommon.sol#L70

```
userBorrowPart[from] += part;
```

- **Medium - TotalBorrow math is correct:** BBLendingCommon.sol#L51-L59

```
function _borrow(
    address from,
    address to,
    uint256 amount,
    uint256 feeAmount
) internal returns (uint256 part, uint256 share) {
    openingFees[to] += feeAmount;

    (totalBorrow, part) = totalBorrow.add(amount + feeAmount, true);
```

  - TODO: After adding `totalBorrow.add(amount)`, the values are still correct

AMT and Shares are correct: BBLendingCommon.sol#L73-L77

```
```solidity
//mint USDO
IUSDOBase(address(asset)).mint(address(this), amount);

//deposit borrowed amount to user
share = _depositAmountToYb(asset, to, assetId, amount);
```

- After receiving the yb asset.
- By withdrawing, we get the exact correct amount.
```

- **Medium - This never reverts, unless the oracle has failed:** BBLendingCommon.sol#L80-L103

```
function _computeVariableOpeningFee(
    uint256 amount
) internal returns (uint256) {
    if (amount == 0) return 0;

    //get asset <> USDC price ( USDO <> USDC )
    (bool updated, uint256 _exchangeRate) = assetOracle.get(oracleData);
    if (!updated) revert OracleCallFailed();

    if (_exchangeRate >= minMintFeeStart) return minMintFee;
    if (_exchangeRate <= maxMintFeeStart) return maxMintFee;

    uint256 fee = maxMintFee -
        (((_exchangeRate - maxMintFeeStart) * (maxMintFee - minMintFee)) /
            (minMintFeeStart - maxMintFeeStart));

    if (fee > maxMintFee) return maxMintFee;
    if (fee < minMintFee) return minMintFee;

    if (fee > 0) {
        return (amount * fee) / FEE_PRECISION;
    }
    return 0;
}
```

- **High - Repay `amount` Math is sound:** BBLendingCommon.sol#L106-L141

```
function _repay(
    address from,
    address to,
    uint256 part
) internal returns (uint256 amountOut) {
    if (part > userBorrowPart[to]) {
        part = userBorrowPart[to];
    }
    if (part == 0) revert NothingToRepay();

    uint256 openingFee = _computeRepayFee(to, part);
    if (openingFee >= part) revert RepayAmountNotValid();

    openingFees[to] -= openingFee;

    uint256 amount;
    (totalBorrow, amount) = totalBorrow.sub(part, true);
    userBorrowPart[to] -= part;

    amountOut = amount;

    yieldBox.withdraw(assetId, from, address(this), amount, 0);

    uint256 accruedFees = amount - part;
    if (accruedFees > 0) {
        uint256 feeAmount = (accruedFees * protocolFee) / FEE_PRECISION;
        amount -= feeAmount;
    }
    uint256 toBurn = (amount - openingFee); //the opening & accrued fees remain in the contract
    //burn USDO
    if (toBurn > 0) {
        IUSDOBase(address(asset)).burn(address(this), toBurn);
    }

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

- If No Interest:
  - For each user, track what they deposited.
  - Have each user repay.
  - Verify that they repay exactly what they deposited.
- If interest: (BASIC):
  - For each user, track what they deposited.
  - Have user user repay.
  - Verify that they repay more than what they deposited.
- If interest: (DIFFICULT):
  - For each user, track what they deposited.
  - On each accrual, recompute each user debt based on the changes in interest.
  - Have user user repay.
  - Verify that they repay exactly that amount when repaying 100%.

Interest Math Side Note

Interest Math could be tested / fuzzed separately.

- Setup a contract with the balances and the interest math (market prob).
- Verify the behaviour given changes in total balances, etc..

This ensures all the underlying math is sound. If you cover all functions here, then as long as you use the same functions to update balances and interests, most properties will be maintained.

Opening Fees:

```
Ms - Never Revert
Never above 100%
Never above amount
```

BBLendingCommon.sol#L143-L163:

```
function _computeRepayFee(
    address user,
    uint256 repayPart
) private view returns (uint256) {
    uint256 _totalPart = userBorrowPart[user];

    if (repayPart == _totalPart) {
        return openingFees[user];
    }

    uint256 _assetDecimals = asset.safeDecimals();
    uint256 repayRatio = _getRatio(repayPart, _totalPart, _assetDecimals);
    // it can return 0 when numerator is very low compared to the denominator
    if (repayRatio == 0) return 0;

    uint256 openingFee = (repayRatio * openingFees[user]) /
        (10 ** _assetDecimals);
    if (openingFee > openingFees[user]) return openingFees[user];

    return openingFee;
}
```

Note: This is wrong math, as this is comparing "amt" to "part": BBLendingCommon.sol#L116-L117

```
uint256 openingFee = _computeRepayFee(to, part);
if (openingFee >= part) revert RepayAmountNotValid();
```

BBLeverage:

- **High - Must be added to check global invariants are held:** BBLeverage.sol#L23-L24

  ```
  function buyCollateral(
  ```

  - Cannot self liquidate.
  - Cannot inflate debt.
  - Cannot over-borrow.
- **Hight - Same as above:** BBLeverage.sol#L83-L84

  ```
  function sellCollateral(
  ```

BBLiquidations:

_Note: You may need to add the concept of System Health/System LTV. After liquidations, the LTV should raise, unless all Positions can be liquidated.

- **High - Liquidations always work when profitable (no reverts)**
- **High - Liquidations never work when an account is not liquidatable**
- **High - Liquidations Math is Sound**
  - No over paying.
  - Properly Paying.
  - Close accounts.
  - No shadow debt.
  - Pays at most User Coll.
  - Pays at least X Premium.
  - Repays 100% of user debt.
- **High - Rounding and repayment:** BBLiquidation.sol#L62-L65

```
(totalBorrow, borrowAmount) = totalBorrow.sub(
    userBorrowPart[user],
    true
);
```

  % value of part doesn't change after a liquidation
- **Medium - Liquidations work when the oracle doesn't** Will require setting up a custom oracle, but generally will work fine as is.
- **High - After refactor - Liquidations can be performed with a lower premium and always work**
- **High - Owner can perform bad debt liquidations, and the system will have it's Health Increased** Since the owner take on the bad debt, the system should be in a better place after this.
- **High - Swap collateral cannot be used to game the system:** BBLiquidation.sol#L71-L77

```
(, uint256 returnedAmount) = _swapCollateralWithAsset(
    collateralShare,
    liquidatorReceiver,
    liquidatorReceiverData
);

asset.safeTransfer(receiver, returnedAmount);
```

  e.g. there's no way to return more than the swapper result. There's no way to steal fees via it.

Market Invariants

- **Critical - Allowances are always paid** If any token is moved, allowances must be paid. This could have many gotchas, but could identify scenarios where tokens can be moved due to rounding down to 0 on the check, in spite of having 0 allowance.

Singularity Invariants

Note: 80% of invariants are the same as BB.

- **Critical - Basket valuation invariant** If basket can be manipulated, then the core invariant is broken.
- **High -** `_getInterestRate` **math is sound**
- **High -** `_getInterestRate` **never overflows nor reverts:** SGLCommon.sol#L37-L38

```
function _getInterestRate()
```

- **Hight - Accounting invariants**

- Shares / Amounts.

**Repay/Withdraw/RemoveAsset - Revert Invariants

```
Should not reverting in most scenarios.
```

Recommendation: Consider writing invariants for SGL first, BB Invariants should be a subset of them.

5.6.25 Inconsistent Usage of Precision Units

Severity: Informational

Context: USDOFlashloanHelper.sol#L19-L20

Description: The variables are inconsistent and may cause issues when:

- Governance has to verify the parameters (due to wrong assumptions).
- An operational mistake is done.
- Reviewers assume all precisions are the same.
- Developers expect precisions to be the same.

This is an incomplete list of all precisions used in the codebase in scope (see USDOFlashloanHelper.sol#L19-L20):

```
uint256 private constant FLASH_MINT_FEE_PRECISION = 1e6;
```

See Market.sol#L99-L100:

```
uint256 internal constant FEE_PRECISION = 1e5;
```

Also see BBStorage.sol#L55-L56:

```
uint256 internal constant DEBT_PRECISION = 1e18;
```

And finally see BaseUSDOStorage.sol#L54-L55:

```
uint256 internal constant SLIPPAGE_PRECISION = 1e4;
```

Recommendation: Consider:

- Documenting explicitly why precisions are different.
- Refactoring to use one "high precision" (1e18) unit, and a "low precision" (1e4 or 1e6) unit.

Tapioca: Creaded PR 295.

5.6.26 Maximum reward tokens in `twTAP` can accidentally be set to break the `rewardTokens` length invariant

Severity: Informational

Context: twTAP.sol#L497-L500, twTAP.sol#L504-L505

Description: `maxRewardTokens` is initialized to `1000`. `addRewardToken()` is used to enforce the invariant for reward token length limit of `rewardTokens.length + 1 <= maxRewardTokens`. However, given that `setMaxRewardTokensLength()` allows the owner to reset it to any value, it can accidentally be set to break the `rewardTokens` length invariant by setting it to a value less that the current `rewardTokens.length + 1`.

Impact: Maximum reward tokens in `twTAP` can accidentally be set to break the `rewardTokens` length invariant.

Likelihood: Very Low + Impact: Very Low = Severity: Informational.

Recommendation: Consider checking that `_length >= rewardTokens.length + 1` in `setMaxRewardTokensLength()`.

Tapioca: Fixed in PR 128.

5.6.27 Mitigation status of Code4rena issues

Severity: Informational

Context: Code4rena report, Code4rena Issues

Description: The protocol underwent a previous security review at Code4rena in July 2023 whose report says:

> The C4 analysis yielded an aggregated total of 159 unique vulnerabilities. Of these vulnerabilities, 60 received a risk rating in the category of HIGH severity and 99 received a risk rating in the category of MEDIUM severity.
>
> Additionally, C4 analysis included 79 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 19 reports recommending gas optimizations.

Since then, the protocol team communicated that they have fixed/acknowledged the reported issues. The current security review besides reviewing additional refactoring/changes since the Code4rena review, is also meant to serve as a mitigation review of the previously reported issues within the codebase in-scope. There were contracts in-scope for Code4rena review that are out-of-scope (OOS) for this review.

The below table tracks Code4rena issues of High & Medium severity, their fix status ("Fixed", "NOT Fixed", "Partially Fixed", "OOS" etc.) based on our assessment at start of our review, references to the current codebase with the fixes/issues, new issues created for C4 issues not/partially fixed and notes.

Issue	Status	References
C4-1695	Fixed	BaseTOFTMarketModule.sol#L88-L93, BaseTOFTMarketDestinationModule.sol#L225-L236
C4-1623	Fixed	TapiocaOptionBroker.sol#L323-L326 and TapiocaOptionBroker.sol#L404-L409
C4-1620	Fixed	
C4-1582	Fixed	
C4-1567	Fixed	
C4-1449	OOS	
C4-1447	OOS	
C4-1432	OOS	
C4-1428	OOS	
C4-1355	Fixed	Now fixed as a part of *`liquidateBadDebt` is not repaying any of the debt, removing assets f
C4-1307	Fixed	BaseTOFTOptionsModule.sol#L53-L60, BaseTOFTOptionsDestinationModule.sol#L73-L76
C4-1293	Fixed	BaseTOFTMarketModule.sol#L90-L93, BaseTOFTMarketDestinationModule.sol#L235-L236
C4-1290	Fixed	BaseTOFTOptionsModule.sol#L89, BaseTOFTOptionsDestinationModule.sol#L176, USDOO
C4-1281	Fixed	BaseTOFTStrategyModule.sol#L50-L57, BaseTOFTStrategyModule.sol#L106-L113, BaseTO
C4-1223	Fixed	USDOFlashloanHelper.sol#L110-L114
C4-1220	Fixed	
C4-1207	Fixed	
C4-1202	Fixed	BaseTOFTLeverageDestinationModule.sol#L170, USDOMarketDestinationModule.sol#L117-
C4-1170	Fixed	twTAP.sol#L83-L85
C4-1168	Fixed	
C4-1165	Fixed	
C4-1164	Fixed	

Issue	Status	References
C4-1163	Fixed	BaseTOFTLeverageDestinationModule.sol#L122-L125, BaseTOFTMarketDestinationModule
C4-1145	Fixed	
C4-1109	Fixed	
C4-1101	Fixed	
C4-1094	Fixed	twTAP.sol#L554-L562
C4-1083	Fixed	BaseTOFTLeverageDestinationModule.sol#L51, BaseTOFTLeverageDestinationModule.sol#
C4-1069	Fixed	BaseTOFT.sol#L625-L666
C4-1057	Fixed	
C4-1046	Fixed	
C4-1043	Fixed	USDOFlashloanHelper.sol#L116-L117
C4-1034	Fixed	BaseTOFTLeverageDestinationModule.sol#L141
C4-1032	Fixed	BaseTOFTStrategyModule.sol#L115-L119
C4-1031	Fixed	BaseTOFTMarketModule.sol#L56-L60
C4-1022	Fixed	twTAP.sol#L157
C4-1021	Fixed	
C4-1019	OOS	
C4-990	OOS	
C4-978	OOS	
C4-943	OOS	
C4-936	OOS	
C4-932	OOS	
C4-918	Fixed	
C4-915	Fixed	
C4-904	Fixed	BaseTOFTLeverageDestinationModule.sol#L149
C4-849	OOS	
C4-836	Fixed	BaseTOFTGenericModule.sol#L280, BaseTOFTOptionsDestinationModule.sol#L176
C4-832	OOS	
C4-674	OOS	
C4-583	Fixed	
C4-541	Fixed	TapOFT.sol#L217
C4-494	OOS	
C4-493	Fixed	SGLLeverage.sol#L1-L122
C4-369	Fixed	Balancer.sol#L410
C4-351	OOS	
C4-329	Fixed	twTAP.sol#L292, twTAP.sol#L425
C4-219	Fixed	BaseTOFT.sol#L554

Issue	Status	References
C4-87	NOT Fixed	Issue *Repayment protocol fees are computed off the protocol inception base and can becom*
C4-41	Fixed	twTAP.sol#L566
C4-1561	NOT Fixed	Issue *ETH market borrow and repay call for the linked BB markets accrual after the state cha*
C4-1553	OOS	
C4-1520	OOS	
C4-1519	OOS	
C4-1504	OOS	
C4-1474	OOS	
C4-1459	OOS	
C4-1456	OOS	
C4-1437	OOS	
C4-1429	OOS	
C4-1425	OOS	
C4-1408	Fixed	
C4-1405	OOS	
C4-1368	Fixed	
C4-1350	Fixed	TapiocaOptionLiquidityProvision.sol#L314-L315
C4-1349	NOT Fixed	Issue *Errors in* `_yieldBoxShares` *accounting will lead to incorrect and potential loss of user f*
C4-1346	Fixed	BaseTOFTStrategyModule.sol#L115-L119, Clickup
C4-1336	OOS	
C4-1333	OOS	
C4-1330	OOS	
C4-1321	OOS	
C4-1300	Fixed	Clickup, LZ airdrop doc
C4-1280	OOS	
C4-1277	Fixed	BigBang.sol#L516, Penrose.sol#L276-L279
C4-1276	Fixed	USDOFlashloanHelper.sol#L119-L160
C4-1264	Fixed	
C4-1248	Fixed	BaseTOFT.sol#L339,BaseTOFTOptionsDestinationModule.sol#L176
C4-1247	Fixed	TapiocaOptionLiquidityProvision.sol#L213
C4-1246	Fixed	TapiocaOptionLiquidityProvision.sol#L36, TapiocaOptionBroker.sol#L312 and many other pla
C4-1241	Fixed	TapOFT.sol#L259-L260
C4-1239	OOS	
C4-1218	Fixed-Reintroduced	TapOFT.sol#L217
C4-1212	Fixed	BaseTOFTMarketDestinationModule.sol#L165-L165
C4-1211	OOS	

Issue	Status	References
C4-1209	OOS	
C4-1184	OOS	
C4-1175	Fixed	AirdropBroker.sol#L492
C4-1174	Fixed	BaseTOFTStrategyDestinationModule.sol#L150-L150
C4-1169	Fixed	BBBorrow.sol#L66, BBLiquidation.sol#L92
C4-1158	Fixed	Penrose.sol#L405, Penrose.sol#L443
C4-1139	Fixed	
C4-1120	NOT Fixed	Issue *New interest protocol fee accounting is incorrectly computed from* `totalBorrow.base`,
C4-1086	NOT Fixed	Issue `SGLLeverage.buyCollateral` *allows any caller to* `withdraw` `supplyShare` *as long as* `c`
C4-1061	Fixed	
C4-1040	Fixed	
C4-1033	Fixed	BaseTOFTMarketDestinationModule.sol#L186, BaseTOFTMarketDestinationModule.sol#L23
C4-1026	NOT Fixed	Issue *Liquidations will fail if oracles revert*, BBLiquidation.sol#L100
C4-1012	NOT Fixed	Issue `collateralizationRate` *can be set to a value that blocks all the liquidations*
C4-1002	Fixed	SGLLiquidation.sol#L89-L119
C4-993	OOS	
C4-989	OOS	
C4-920	Fixed	
C4-916	OOS	
C4-894	Fixed	USDOLeverageDestinationModule.sol#L71-L82, USDOMarketDestinationModule.sol#L81-L9
C4-889	OOS	
C4-879	Partially Fixed	Issue *Airdropped options cannot be exercised with tokens that have greater than 18 decimals*
C4-813	Fixed	Balancer.sol#L372, Balancer.sol#L398
C4-805	OOS	
C4-758	Fixed	twTAP.sol#L390-L393
C4-743	Invalid	
C4-700	Fixed	SGLLendingCommon.sol#L82-L84
C4-569	Fixed	TapiocaOptionBroker.sol#L102, TapiocaOptionBroker.sol#L356-L360
C4-568	OOS	
C4-561	OOS	
C4-483	Partially Fixed	Issue *Missing* `unchecked` *causes* `FullMath.mulDiv()` *to revert instead of overflowing as exp*
C4-476	OOS	
C4-385	OOS	
C4-378	OOS	
C4-377	Fixed	BaseTOFTLeverageModule.sol#L60-L65,BaseTOFTLeverageDestinationModule.sol#145, U
C4-365	OOS	

Issue	Status	References
C4-349	NOT Fixed	Issue *`oTAP.participate()` will always revert if `msg.sender` is approved but not owner*
C4-337	Fixed	
C4-336	Fixed	Balancer.sol#L398-L412
C4-334	Fixed	Balancer.sol#L213
C4-323	Fixed	TapiocaOptionBroker.sol#L507-L512
C4-285	OOS	
C4-283	OOS	
C4-250	OOS	
C4-242	Fixed	
C4-232	Acknowledged	
C4-209	OOS	
C4-201	Fixed	USDOOptionsDestinationModule.sol#L145-L155, BaseTOFTOptionsDestinationModule.sol#L
C4-200	Fixed	
C4-189	Partially Fixed	Issue *It is possible to exercise TAP option an extra time compared to lock duration*
C4-188	Acknowledged	
C4-187	Partially Fixed	Issue *Expiry overflow allows attacker to claim majority of rewards and gain voting power whil*
C4-175	OOS	
C4-163	OOS	
C4-162	OOS	
C4-145	Fixed	BBLiquidation.sol#L29-L78
C4-128	Fixed	BaseTOFTStrategyDestinationModule.sol#L75-L86, BaseTOFTMarketDestinationModule.sol
C4-64	Fixed	BBLendingCommon.sol#L111-L113
C4-31	Fixed	BaseTapOFT.sol#L268-L280
C4-27	Partially Fixed	Issue *Unchecked revert message lengths may lead to protocol-wide DoS*, Penrose.sol#L499
C4-26	Fixed	BaseTapOFT.sol#L164

5.6.28 Joint informational/QA nitpicks

Severity: Informational

Context: Global scope

Description: Below is a list of all the informational/nitpicks in the codebases with recommendations categorized. There are temporarily categorized into respective PR's.

```
ChangeLog: 26/01/2024
```

- TapiocaOptionLiquidityProvision.sol#L279: Typo: `function setSGLPoolWeight`.
- TapOFT.sol#L232: Conversion isn't needed, `_computeEmission` already returns `uint256`.
- Vesting.sol#L188: Missing emit of `UserRegistered` event as done in `registerUser()`.

- Vesting.sol#L128: Gas: Can remove `seeded == 0` from the conditional expression because both `start` and `seeded` are initialized to non-zero values in init() and if `start != 0` then `seeded != 0` is guaranteed.
- BaseTOFTLeverageDestinationModule.sol#L98: Gas: Cache `_sd2ld(amountSD)` instead of calling it multiple times (L79, L87, L99, L109).
- Balancer.sol#L221: Suggest moving this before the send's above to comply with CEI.

`ChangeLog:25/01/2024`

- Singularity.sol#L282-L284: Seems to remove assets from `from`, not `msg.sender`.
- SGLLendingCommon.sol#L114, SGLLendingCommon.sol#L75, SGLCommon.sol#L258: Consider using OZ `SafeCast` lib when casting down to avoid potential overflows.
- SGLLiquidation.sol#L56, BBLiquidation.sol#L54: Nitpick: not a `require` statement.
- Singularity.sol#L137-L141: Consider adding zero-address checks.
- Singularity.sol#L246: Is this really meant for `i++`? If so, can move it inside the loop.
- Singularity.sol#L203: Notice that `amount` here is in base units (ones as of system inception), i.e. it's not current token amount. Consider highlighting this in description.
- USDOLeverageDestinationModule.sol#L86: Consider moving this helper function to `USDOCommon.sol` given that this logic is used in `*DestinationModules`.
- USDOFlashloanHelper.sol#L105-L108: Add a sanity check upfront that `token == usdo` instead of this being done within `flashFee()` downstream.
- USDOFlashloanHelper.sol#L160: Can add `if (allowance(address(receiver), address(this)) < fee)` check earlier at fee calculation to ensure success of this `transferFrom`.
- Penrose.sol#L247: Can set this as immutable in the constructor instead of passing as a parameter.
- Penrose.sol#L277: Given the default value of `5e15`, recommend enforcing a reasonable upper threshold to prevent footguns.
- twTAP.sol#L479: Consider adding a sanity check for `_rewardTokenId != 0` to skip the reserved `0x0` address at index 0.
- twTAP.sol#L480: Consider checking for a DBZ for `totals.netActiveVotes`.
- twTAP.sol#L490: Consider moving this sanity check to the beginning of the function.
- twTAP.sol#L222: QA: This would be "`createdAt`".
- twTAP.sol#L226: This would be "`expiresAt`".

`ChangeLog: 22/12/2023`

- **Singularity contracts**
  - SGLBorrow.sol#L93: This appears to be the same as BoringERC20.sol#L50-L56. Why not use that library function instead?
  - SGLCollateral.sol#L23: Missing `@param` for `amount`.
- **bigBang Contracts**
  - BBCollateral.sol#L13: This should be `from` instead of `msg.sender`.
  - BBCollateral.sol#L18: Missing `@param` for `amount`
  - BBCommon.sol#L44: Naming: consider changing `debt` to `debtRate`.

- BBCommon.sol#L110: Nitpick: Tokens are transferred from `from` not `msg.sender`.
- BBCommon.sol#L128: Missing NatSpec.
- BBLendingCommon.sol#L61: Can be removed - the following conditional checks should make this redundant.
- BBLendingCommon.sol#L160: `openingFees[user]` can be saved to memory to save SLOADs.
- BBLiquidation.sol#L54: Nitpick: Stale comment from before the `require` was converted to a custom error.
- BigBang.sol#L44: Nitpick: Should be "BigBang" not "Singularity".
- BigBang.sol#L135: Consider adding zero-address checks.
- BigBang.sol#L216: Nitpick: "BigBang" not "BingBang".
- BigBang.sol#L248: Missing NatSpec `@param` for `amount`.
- BigBang.sol#L378: Missing NatSpec comments here.
- BigBang.sol#L399: Missing NatSpec `@params`.
- BigBang.sol#L485: Missing @return.
- BigBang.sol#L506: Missing NatSpec.
- BigBang.sol#L525: Suggest using `MaxDebtRateNotValid` instead, as on L148.

`Changelog: 12/12/2023`

- **Market contracts**
  - Market.sol#L124: Suggest adding the oracle address as parameter.
  - Market.sol#L186: Missing `@params`.
  - Market.sol#L187: Suggest emitting missing events for market config parameters: `_callerFee`, `_protocolFee`, `_liquidationBonusAmount`, `_minLiquidatorReward`, `maxLiquidatorReward`, `_collateralizationRate` and `_liquidationCollateralizationRate`.
  - Market.sol#L279: Missing `@param` for type.
  - Market.sol#L290: Missing NatSpec.
  - Market.sol#L329: Missing `@return` for minTVL & maxTVL.
  - Market.sol#L386: This should be `@param` not `@notice`.
  - Market.sol#L525: Nitpick: Should be `maxBorrowable`.
  - MarketERC20.sol#L224: QA: Using an Enum can help with clarity.
  - MarketERC20.sol#L296: QA: Technically you don't need both options as you have `actionType`. Having both typehashes is overcomplicated, although not a major risk.
  - MarketLiquidatorReceiver.sol#L35: Missing NatSpec.
- **USDO Modules**
  - USDOGenericModule.sol#L21: Missing NatSpec on all functions here.
  - USDOMarketModule.sol#L14: Nitpick: Remove stale comment and commented import above re: Rebase.
  - USDOMarketModule.sol#L21: Remove unused custom error.
  - USDOMarketModule.sol#L29: Missing NatSpec on all functions.

- **USDO Contracts**
  - BaseUSDO.sol#L96: Missing zero checks.
  - BaseUSDO.sol#L169: Nitpick: We can declare a different custom error for zero-address instead of overloading `NotAuthorized`.
  - BaseUSDO.sol#L264: Missing `@param` for `zroPaymentAddress`.
  - BaseUSDO.sol#L119: Nitpick: Can avoid use of `token` in `maxFlashLoan()` and use `usdo` instead of `token` in `flashFee()`.
- **Penrose.sol**
  - Penrose.sol#L57: Set only in constructor - can be immutable.
  - Penrose.sol#L156: Unused event - missing functionality? *Tapioca Team: we removed the Swapper from Penrose at some point.*
  - Penrose.sol#L169: Consider following the convention like other places by using `event BigBangEthMarketUpdated(address indexed _oldAddress, address indexed _newAddress);`
  - Penrose.sol#L171: Consider following the convention like other places by using `event BigBangEthMarketDebtRateUpdated(uint256 indexed _oldRate, uint256 indexed _newRate);`
  - Penrose.sol#L530: Consider renaming this function to `getAllMasterContractClones()` because:
    - This does not return the length.
    - This enumerates and returns all clones of all master contracts.
    - This is a public function and hence shouldn't use the leading _ naming convention.
  - Penrose.sol#L245: Is this meant to be `onlyOwner`? Probably - if so, this can be moved to the `OWNER FUNCTIONS` section below, instead of `PUBLIC FUNCTIONS`.
  - Penrose.sol#L242: Nitpick: This should be "call `_depositFeesToTwTap()`".
  - Penrose.sol#L268: For all address setters, consider adding zero-address sanity checks.
  - Penrose.sol#L249: Use `ZeroAddress()` instead.
  - Penrose.sol#L302: Use `ZeroAddress()` instead.
  - Penrose.sol#L282: Missing `@param` for `_market`.
  - Penrose.sol#L376: Missing `@return`.
  - Penrose.sol#L377: Nitpick: Consider renaming to `deployAndRegisterSingularity` for sync with logic.
  - Penrose.sol#L398: Nitpick: Consider renaming to `registerSingularity` for sync with logic.
  - Penrose.sol#L397: Missing `@param` for `_contract`.
  - Penrose.sol#L413: Missing `@return`.
  - Penrose.sol#L41: Nitpick: Consider renaming to `deployAndRegisterBigBang` for sync with logic.
  - Penrose.sol#L435: Missing `@param` for `_contract`.
  - Penrose.sol#L436: Nitpick: Consider renaming to `registerBigBang` for sync with logic.
  - Penrose.sol#L447: Missing NatSpec.
- **governance/twTap.sol**
  - twTAP.sol#L143: Nitpick: Should be `CannotClaim`.
  - twTAP.sol#L284: Nitpick: Should be twTAP position.
- **Option-Airdorp**
  - AirdropBroker.sol#L307: Incorrect copy-paste comment from `TapiocaOptionBroker.newEpoch()`.

- AirdropBroker.sol#L355: Recommend adding an event emit for all privileged functions.
- AirdropBroker.sol#L358: Nitpick: Should be `registerUsersForPhase` because this is registering more than one user.
- AirdropBroker.sol#L475: Incorrect comments: This is Phase 3 and calldata is PCNFT tokenID.
- AirdropBroker.sol#L498: This does not implement the five sub-phases of Phase-4 as described in the Docs - suggest to sync up Docs with implementation.

- **TapiocaZ**
  - Link: This is technically `calldata` and not `bytecode`.

Recommendation: Address the aforementioned nitpicks.

Tapioca: Ongoing fixes:

- **tapioca-bar:** Most fixes in PR 258.
- natspec for `MarketLiquidatorReceiver` in PR 267.
- Penrose immutable type for host chain in PR 265.
- **tap-token:** in PR 119

5.6.29 `Penrose.executeMarketFn` pause may not be desired

Severity: Informational

Context: Penrose.sol#L448-L457

Description: `conservator` is `!= owner`, so in some cases this can prevent changing a setting as you won't be able to change it without unpausing (see Penrose.sol#L448-L457):

```
function executeMarketFn(
    address[] calldata mc,
    bytes[] memory data,
    bool forceSuccess
)
    external
    onlyOwner
    notPaused /// @audit QA: You prob may want this to still work while paused for admin purposes
    returns (bool[] memory success, bytes[] memory result) /// You can multicall to not refactor
{
```

Both options are not desirable, but it's worth keeping in mind that if you ever need to change a setter while paused you'll have to:

1. Make owner the conservator.
2. Make a multicall via owner (assuming owner is going to be a Safe or a Timelock).
3. Unpase → do the operation → Pause again.
4. Any other operative procedure could cause further issues.

Recommendation: Consider allowing the owner to unpause as well, or keep this in mind as an operative risk.

Tapioca: Allowed owner to execute actions on market when paused in PR 263.

Spearbit: Verified.