Malt Protocol - Versus contest Findings & Analysis Report

2023-05-01

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Malt Protocol smart contract system written in Solidity. The audit contest took place between February 14—February 20 2023.

Wardens

In Code4rena's Versus contests, the competition is limited to a small group of wardens; for this contest 4 Wardens contributed reports:

1. KingNFT
2. cccz
3. hansfriese
4. minhquanym

This contest was judged by Picodes.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 22 unique vulnerabilities. Of these vulnerabilities, 6 received a risk rating in the category of HIGH severity and 16 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 4 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 3 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Malt Protocol contest repository, and is composed of 11 smart contracts written in the Solidity programming language and includes 2,617 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](), specifically our section on [Severity Categorization]().

High Risk Findings (6)

[H-01] `RewardThrottle.checkRewardUnderflow()` might track the cumulative `APR` s wrongly.

Submitted by [hansfriese]()

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/RewardSystem/RewardThrottle.sol#L445-L455

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/RewardSystem/RewardThrottle.sol#L576

Impact

`RewardThrottle.checkRewardUnderflow()` might calculate the cumulative `APR` s for epochs wrongly.

As a result, `cashflowAverageApr` will be calculated incorrectly in `updateDesiredAPR()` , and `targetAPR` would be changed unexpectedly.

Proof of Concept

In `checkRewardUnderflow()`, it calls a `_sendToDistributor()` function to update cumulative `APR`s after requesting some capitals from the overflow pool.

```
File: 2023-02-malt\contracts\RewardSystem\RewardThrottle.sol
if (epoch > _activeEpoch) {
  for (uint256 i = _activeEpoch; i < epoch; ++i) {
    uint256 underflow = _getRewardUnderflow(i);

    if (underflow > 0) {
      uint256 balance = overflowPool.requestCapital(unc

      _sendToDistributor(balance, i);   //@audit cumulat
    }
  }
}
```

The main reason for this issue is that `_sendToDistributor()` doesn't update the cumulative `APR`s when `amount == 0` and the below scenario would be possible.

1. Let's assume `activeEpoch = 100` and `epoch = 103`. It's possible if the active epoch wasn't updated for 2 epochs.
2. After that, the `checkRewardUnderflow()` function will call `_fillInEpochGaps()` and the cumulative `APR`s will be settled accordingly.
3. And it will try to request capitals from the `overflowPool` and increase the rewards for epochs.
4. At epoch 100, it requests some positive `balance` from `overflowPool` and increases the cumulative `APR`s for epoch 101 correctly in `_sendToDistributor()`.

```
File: 2023-02-malt\contracts\RewardSystem\RewardThrottle.sol
state[epoch].rewarded = state[epoch].rewarded + rewarde
state[epoch + 1].cumulativeCashflowApr =
  state[epoch].cumulativeCashflowApr +
  epochCashflowAPR(epoch);
state[epoch + 1].cumulativeApr =
  state[epoch].cumulativeApr +
617:       epochAPR(epoch);
618:     state[epoch].bondedValue = bonding.averageBondedValue(e
```

5. After that, the `overflowPool` doesn't have any remaining funds and the `balance(At L450)` will be 0 for epochs 101, 102.
6. So `_sendToDistributor()` will be terminated right away and won't increase the cumulative `APR` s of epoch 102 according to epoch 101 and this value won't be changed anymore because the `activeEpoch` is 103 already.

```
File: 2023-02-malt\contracts\RewardSystem\RewardThrottle.sol
575:   function _sendToDistributor(uint256 amount, uint256 epoch
576:     if (amount == 0) {
577:       return;
578:     }
```

As a result, the cumulative `APR` s will save smaller values from epoch 102 and `cashflowAverageApr` will be smaller also if the `smoothingPeriod` contains such epochs in `updateDesiredAPR()` .

```
File: 2023-02-malt\contracts\RewardSystem\RewardThrottle.sol
139:     uint256 cashflowAverageApr = averageCashflowAPR(smoothi
```

So the `updateDesiredAPR()` function will change the `targetAPR` using the smaller average value and the smoothing logic wouldn't work as expected.

Recommended Mitigation Steps

I think `_sendToDistributor()` should update the cumulative `APR` s as well when `amount == 0` .

```
  function _sendToDistributor(uint256 amount, uint256 epoch) int
    if (amount == 0) {
        state[epoch + 1].cumulativeCashflowApr = state[epoch].cu
        state[epoch + 1].cumulativeApr = state[epoch].cumulative
        state[epoch].bondedValue = bonding.averageBondedValue(ep
        return;
    }
```

[0xScotch (Malt) confirmed and commented](#):

> Interesting finding. It’s valid but the bug would actually result in the protocol retaining more capital due to reporting lower APRs than it should.

[H-02] RewardThrottle: If an epoch does not have any profit, then there may not be rewards for that epoch at the start of the next epoch.

Submitted by [cccz](#), also found by [hansfriese](#)

In RewardThrottle, both checkRewardUnderflow and fillInEpochGaps call `\_fillInEpochGaps` to fill the state of the previous epoch without profit, the difference being that checkRewardUnderflow will request the reward from the overflowPool and distribute the reward, whereas fillInEpochGaps does not.

```
function checkRewardUnderflow() public onlyActive {
  uint256 epoch = timekeeper.epoch();

  uint256 _activeEpoch = activeEpoch; // gas

  // Fill in gaps so we have a fresh foundation to calculate f
  _fillInEpochGaps(epoch);

  if (epoch > _activeEpoch) {
    for (uint256 i = _activeEpoch; i < epoch; ++i) {
      uint256 underflow = _getRewardUnderflow(i);

      if (underflow > 0) {
        uint256 balance = overflowPool.requestCapital(underflo

        _sendToDistributor(balance, i);
      }
    }
  }
}
```

```
function fillInEpochGaps() external {
  uint256 epoch = timekeeper.epoch();

  _fillInEpochGaps(epoch);
}
```

This results in that when an epoch does not have any profit, then at the start of the next epoch that epoch will have a reward if checkRewardUnderflow is called, and no reward if `fillInEpochGaps` is called.

According to the documentation, when an epoch is not profitable enough, the reward should be requested from the overflowPool, so checkRewardUnderflow should be called. And if `fillInEpochGaps` is called first, the epoch will lose its reward.

Note: populateFromPreviousThrottle will also cause epochs without any profit to lose their rewards

```
function populateFromPreviousThrottle(address previousThrottle
  external
  onlyRoleMalt(ADMIN_ROLE, "Only admin role")
{
  RewardThrottle previous = RewardThrottle(previousThrottle);
  uint256 _activeEpoch = activeEpoch; // gas

  for (uint256 i = _activeEpoch; i < epoch; ++i) {
    (
      uint256 profit,
      uint256 rewarded,
      uint256 bondedValue,
      uint256 desiredAPR,
      uint256 epochsPerYear,
      uint256 cumulativeCashflowApr,
      uint256 cumulativeApr
    ) = previous.epochData(i);

    state[i].bondedValue = bondedValue;
    state[i].profit = profit;
    state[i].rewarded = rewarded;
    state[i].epochsPerYear = epochsPerYear;
    state[i].desiredAPR = desiredAPR;
    state[i].cumulativeCashflowApr = cumulativeCashflowApr;
```

```
            state[i].cumulativeApr = cumulativeApr;
          }

          activeEpoch = epoch;
        }
```

Proof of Concept

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/RewardSystem/RewardThrottle.sol#L437-L462

Recommended Mitigation Steps

Consider removing the `fillInEpochGaps` function, or only allowing it to be called when the contract is not active.

0xScotch (Malt) confirmed and commented:

> We will be removing both implementations of `fillInEpochGaps`.

[H-03] Manipulation of `livePrice` to receive `defaultIncentive` in 2 consecutive blocks

Submitted by **minhquanym**

In StabilizerNode, the default behaviour when twap is below the lower peg threshold, all transfers to the amm pool are blocked. However when `usePrimedWindow = true`, it will only block transfers for `primedWindow = 10` blocks. After 10 blocks, the block automatically stops and allows free market trading.

The first call to start this priming will receive `defaultIncentive` Malt and set `primedBlock` to start the priming. However, function `_validateSwingTraderTrigger()` which is used to validate and start the priming using `livePrice` is easy to be manipulated. Attacker can manipulate it to receive `defaultIncentive` in 2 consecutive blocks.

Proof of Concept

Consider the scenario:

1. Block i, twap is below the value returned from `maltDataLab.getSwingTraderEntryPrice()`, attacker call `stabilize()` and receive `defaultIncentive`. `primedBlock = block.number`.
2. Block i+1, call to `_validateSwingTraderTrigger()` return `true` and trigger swing trader to bring the price back to peg. It's also reset `primedBlock = 0` (stop blocking transfer to AMM pool)
3. Since only 1 block pass, let's assume twap is still below the value returned from `maltDataLab.getSwingTraderEntryPrice()` (because twap moves slowly and will not change immediately to current price)
4. Now attacker can use flash loan to manipulate the `livePrice` to be larger than `entryPrice` (tranfer to AMM is not blocked) and call `stabilize()` to receive incentive again then repay the flash loan.

Attacker cost is only flash loan fee, since his call will start an auction but not trigger swing trader so the state of AMM pool when he repays the flash loan is still the same (only added flash loan fee).

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L312-L334

```
  function _validateSwingTraderTrigger(uint256 livePrice, uint256
      internal
      returns (bool)
    {
      if (usePrimedWindow) {
        if (livePrice > entryPrice) {
          return false;
        }

        if (block.number > primedBlock + primedWindow) {
          primedBlock = block.number;
          malt.mint(msg.sender, defaultIncentive * (10**malt.decim
          emit MintMalt(defaultIncentive * (10**malt.decimals()));
          return false;
        }
        if (primedBlock == block.number) {
          return false;
        }
      }

      return true;
    }
```

Recommended Mitigation Steps

Consider not giving incentives for caller or reset the `primedBlock` at least after `primedWindow` blocks.

0xScotch (Malt) commented:

> I'm kinda skeptical of this but I think its possible in theory.

> However:

- `stabilize` can only be called via EOA due to `msg.sender == tx.origin` check (in `onlyEOA` modifier)
- Size of flashloan required is proportional to the size of the pool (as you have to manipulate price of that pool) while the incentive is fixed. So it seems like this would quickly become unprofitable

> I would be very curious to see a real PoC of this rather than just a theoretical threat.

Picodes (judge) commented:

> Regarding the previous comment:

- The `onlyEOA` check can be bypassed using a sandwich attack instead of a flashloan so the possibility of a MEV attack still exists
- We should consider that the cost of capital within a block is 0. For example, Euler already proposes feeless flashloans of up to their TVL. See https://twitter.com/euler_mab/status/1595725665868910595. However there would still be the cost of using the AMM to manipulate the price.

Picodes (judge) commented:

> Although the possibility of this being implemented depends on the size of the incentives and the cost of manipulating the AMM, it does not seem so unlikely. It could lead to a significant loss for the protocol, so I agree that high severity is appropriate.

0xScotch (Malt) acknowledged

[H-04] SwingTraderManager.addSwingTrader will push traderId with `active = false` to activeTraders

Submitted by cccz, also found by hansfriese

In SwingTraderManager.addSwingTrader, if `active = false`, the traderId is also pushed to activeTraders.

```
  function addSwingTrader(
    uint256 traderId,
    address _swingTrader,
    bool active,
    string calldata name
  ) external onlyRoleMalt(ADMIN_ROLE, "Must have admin privs") {
    SwingTraderData storage trader = swingTraders[traderId];
    require(traderId > 2 && trader.id == 0, "TraderId already us
    require(_swingTrader != address(0), "addr(0)");

    swingTraders[traderId] = SwingTraderData({
      id: traderId,
      index: activeTraders.length,
      traderContract: _swingTrader,
      name: name,
      active: active
    });

    activeTraders.push(traderId);

    emit AddSwingTrader(traderId, name, active, _swingTrader);
  }
```

Afterwards, if toggleTraderActive is called on the traderId, the traderId will be pushed to activeTraders again.

```
function toggleTraderActive(uint256 traderId)
  external
  onlyRoleMalt(ADMIN_ROLE, "Must have admin privs")
{
  SwingTraderData storage trader = swingTraders[traderId];
  require(trader.id == traderId, "Unknown trader");

  bool active = !trader.active;
  trader.active = active;

  if (active) {
    // setting it to active so add to activeTraders
    trader.index = activeTraders.length;
    activeTraders.push(traderId);
  } else {
```

This means that in `getTokenBalances()/calculateSwingTraderMaltRatio()`, since there are two identical traderIds in activeTraders, the data in this trader will be calculated twice.

Wrong `getTokenBalances()` will result in wrong data when `syncGlobalCollateral()`.

```
function getTokenBalances()
  external
  view
  returns (uint256 maltBalance, uint256 collateralBalance)
{
  uint256[] memory traderIds = activeTraders;
  uint256 length = traderIds.length;

  for (uint256 i; i < length; ++i) {
    SwingTraderData memory trader = swingTraders[activeTraders
    maltBalance += malt.balanceOf(trader.traderContract);
    collateralBalance += collateralToken.balanceOf(trader.trac
  }
}
```

Wrong `calculateSwingTraderMaltRatio()` will cause `MaltDataLab.getRealBurnBudget()/getSwingTraderEntryPrice()` to be wrong.

```
function calculateSwingTraderMaltRatio()
  public
  view
  returns (uint256 maltRatio)
{
  uint256[] memory traderIds = activeTraders;
  uint256 length = traderIds.length;
  uint256 decimals = collateralToken.decimals();
  uint256 maltDecimals = malt.decimals();
  uint256 totalMaltBalance;
  uint256 totalCollateralBalance;

  for (uint256 i; i < length; ++i) {
    SwingTraderData memory trader = swingTraders[activeTraders
    totalMaltBalance += malt.balanceOf(trader.traderContract);
    totalCollateralBalance += collateralToken.balanceOf(
      trader.traderContract
    );
  }

  totalMaltBalance = maltDataLab.maltToRewardDecimals(totalMal

  uint256 stMaltValue = ((totalMaltBalance * maltDataLab.price
    (10**decimals));

  uint256 netBalance = totalCollateralBalance + stMaltValue;

  if (netBalance > 0) {
    maltRatio = ((stMaltValue * (10**decimals)) / netBalance);
  } else {
    maltRatio = 0;
  }
}
```

What’s more serious is that even if toggleTraderActive is called again, only one traderId will pop up from activeTraders, and the other traderId cannot be popped up.

```
  } else {
    // Becoming inactive so remove from activePools
    uint256 index = trader.index;
    uint256 lastTrader = activeTraders[activeTraders.length -

    activeTraders[index] = lastTrader;
    activeTraders.pop();

    swingTraders[lastTrader].index = index;
    trader.index = 0;
  }
```

This causes the trade to participate in the calculation of `getTokenBalances()/calculateSwingTraderMaltRatio()` even if the trade is deactive.

Considering that the active parameter is likely to be false when addSwingTrader is called and cannot be recovered, this vulnerability should be High risk.

Proof of Concept

```
function testAddSwingTrader(address newSwingTrader) public {
  _setupContract();
  vm.assume(newSwingTrader != address(0));
  vm.prank(admin);
  swingTraderManager.addSwingTrader(3, newSwingTrader, false,

  (
    uint256 id,
    uint256 index,
    address traderContract,
    string memory name,
    bool active
  ) = swingTraderManager.swingTraders(3);

  assertEq(id, 3);
  assertEq(index, 2);
  assertEq(traderContract, newSwingTrader);
  assertEq(name, "Test");
  assertEq(active, false);
  vm.prank(admin);
  swingTraderManager.toggleTraderActive(3);
  assertEq(swingTraderManager.activeTraders(2),3);
```

```
    assertEq(swingTraderManager.activeTraders(3),3); // @audit:a
    vm.prank(admin);
    swingTraderManager.toggleTraderActive(3);
    assertEq(swingTraderManager.activeTraders(2),3);
  }
```

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/SwingTraderManager.sol#L397-L447

Recommended Mitigation Steps

Change to:

```
  function addSwingTrader(
    uint256 traderId,
    address _swingTrader,
    bool active,
    string calldata name
  ) external onlyRoleMalt(ADMIN_ROLE, "Must have admin privs") {
    SwingTraderData storage trader = swingTraders[traderId];
    require(traderId > 2 && trader.id == 0, "TraderId already us
    require(_swingTrader != address(0), "addr(0)");

    swingTraders[traderId] = SwingTraderData({
      id: traderId,
-     index: activeTraders.length,
+     index: active ? activeTraders.length : 0,
      traderContract: _swingTrader,
      name: name,
      active: active
    });
+  if(active) activeTraders.push(traderId);

-   activeTraders.push(traderId);

    emit AddSwingTrader(traderId, name, active, _swingTrader);
  }
```

[0xScotch (Malt) confirmed]()

[H-05] `_distributeProfit` will use the stale `globalIC.swingTraderCollateralDeficit()/swingTraderCollateralRatio()`, which will result in incorrect profit distribution

Submitted by cccz

The `\_distributeProfit()` (called by handleProfit()) will use `globalIC.swingTraderCollateralDeficit()/swingTraderCollateralRatio()` when distributing profits, and the latest `globalIC.swingTraderCollateralDeficit()/swingTraderCollateralRatio()` needs to be used to ensure that profits are distributed correctly.

```
    uint256 globalSwingTraderDeficit = (maltDataLab.maltToRewar
      globalIC.swingTraderCollateralDeficit()
    ) * maltDataLab.priceTarget()) / (10**collateralToken.decima

    // this is already in collateralToken.decimals()
    uint256 lpCut;
    uint256 swingTraderCut;

    if (globalSwingTraderDeficit == 0) {
      lpCut = distributeCut;
    } else {
      uint256 runwayDeficit = rewardThrottle.runwayDeficit();

      if (runwayDeficit == 0) {
        swingTraderCut = distributeCut;
      } else {
        uint256 totalDeficit = runwayDeficit + globalSwingTrader
```

However, the two calls to handleProfit in the contract do not call syncGlobalCollateral to synchronize the data in globalIC.

syncGlobalCollateral will use the data in `getCollateralizedMalt()`, including the collateralToken balance in overflowPool/swingTraderManager/liquidityExtension and the malt balance in swingTraderManager.

```
function syncGlobalCollateral() public onlyActive {
  globalIC.sync(getCollateralizedMalt());
}

function getCollateralizedMalt() public view returns (PoolColl
  uint256 target = maltDataLab.priceTarget(); // 是否选用  getA

  uint256 unity = 10**collateralToken.decimals();

  // Convert all balances to be denominated in units of Malt t
  uint256 overflowBalance = maltDataLab.rewardToMaltDecimals((
    address(overflowPool)
  ) * unity) / target);
  uint256 liquidityExtensionBalance = (collateralToken.balance
    address(liquidityExtension)
  ) * unity) / target;
  (
    uint256 swingTraderMaltBalance,
    uint256 swingTraderBalance
  ) = swingTraderManager.getTokenBalances();
  swingTraderBalance = (swingTraderBalance * unity) / target;
```

1. Before handleProfit is called by StabilizerNode.stabilize.

```
profitDistributor.handleProfit(rewards);
```

a. checkAuctionFinalization is called to liquidityExtension.allocateBurnBudget, which transfers the collateralToken from liquidityExtension to swingTrader. The increase of collateralToken in swingTrader will make the data in globalIC stale.

```
function allocateBurnBudget(uint256 amount)
    external
    onlyRoleMalt(AUCTION_ROLE, "Must have auction privs")
    onlyActive
    returns (uint256 purchased)
  {
    // Send the burnable amount to the swing trader so it can be
    require(
      collateralToken.balanceOf(address(this)) >= amount,
      "LE: Insufficient balance"
    );
    collateralToken.safeTransfer(address(swingTrader), amount);

    emit AllocateBurnBudget(amount);
  }
```

b. swingTraderManager.sellMalt will exchange malt for collateralToken, and the increase of collateralToken in swingTrader will also make the data in globalIC stale.

```
  uint256 swingAmount = swingTraderManager.sellMalt(tradeSize)
```

2. Before SwingTrader.sellMalt is called to handleProfit.

```
function _handleProfitDistribution(uint256 profit) internal vi
  if (profit != 0) {
    collateralToken.safeTransfer(address(profitDistributor), p
    profitDistributor.handleProfit(profit);
  }
}
```

a. dexHandler.sellMalt will exchange malt for collateralToken, and the increase of collateralToken in swingTrader will also make the data in globalIC stale.

```
  malt.safeTransfer(address(dexHandler), maxAmount);
  uint256 rewards = dexHandler.sellMalt(maxAmount, 10000);
```

One obvious effect is that as the collateralToken in swingTrader increases, collateral.swingTrade will be smaller than it actually is, and the result of `globalIC.swingTraderCollateralDeficit()` will be larger than it should be.

```
function swingTraderCollateralDeficit() public view returns (u
  // Note that collateral.swingTrader is already denominated i
  uint256 maltSupply = malt.totalSupply();
  uint256 collateral = collateral.swingTrader; // gas

  if (collateral >= maltSupply) {
    return 0;
  }

  return maltSupply - collateral;
}
```

thus making lpCut larger:

```
uint256 globalSwingTraderDeficit = (maltDataLab.maltToRewar
  globalIC.swingTraderCollateralDeficit()
) * maltDataLab.priceTarget()) / (10**collateralToken.decima

// this is already in collateralToken.decimals()
uint256 lpCut;
uint256 swingTraderCut;

if (globalSwingTraderDeficit == 0) {
  lpCut = distributeCut;
} else {
  uint256 runwayDeficit = rewardThrottle.runwayDeficit();

  if (runwayDeficit == 0) {
    swingTraderCut = distributeCut;
  } else {
    uint256 totalDeficit = runwayDeficit + globalSwingTrade

    uint256 globalSwingTraderRatio = maltDataLab.maltToRewa
      globalIC.swingTraderCollateralRatio()
    );

    // Already in collateralToken.decimals
    uint256 poolSwingTraderRatio = impliedCollateralService
      .swingTraderCollateralRatio();

    if (poolSwingTraderRatio < globalSwingTraderRatio) {
      swingTraderCut = (distributeCut * swingTraderPreferen
      lpCut = distributeCut - swingTraderCut;
    } else {
      lpCut =
        (((distributeCut * runwayDeficit) / totalDeficit) *
          (10000 - lpThrottleBps)) /
        10000;
```

Proof of Concept

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/ProfitDistributor.sol#L164-L184

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L423-L424

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/SwingTrader.sol#L176-L181

Recommended Mitigation Steps

Call syncGlobalCollateral to synchronize the data in globalIC before calling handleProfit.

0xScotch (Malt) confirmed

[H-06] StabilizerNode.stabilize uses stale GlobalImpliedCollateralService data, which will make stabilize incorrect

Submitted by cccz

In StabilizerNode.stabilize, `impliedCollateralService.syncGlobalCollateral()` is called only at the end of the function to synchronize the GlobalImpliedCollateralService data.

```
        if (!_shouldAdjustSupply(exchangeRate, stabilizeToPeg)) {
          lastStabilize = block.timestamp;
          impliedCollateralService.syncGlobalCollateral();
          return;
        }
    ...
        if (trackAfterStabilize) {
          maltDataLab.trackPool();
        }
        impliedCollateralService.syncGlobalCollateral();
        lastStabilize = block.timestamp;
      }
```

syncGlobalCollateral will use the data in `getCollateralizedMalt()` , which includes the collateralToken balance in overflowPool/swingTraderManager/liquidityExtension and the malt balance in swingTraderManager.

```
      function syncGlobalCollateral() public onlyActive {
        globalIC.sync(getCollateralizedMalt());
      }
    ...
      function getCollateralizedMalt() public view returns (PoolColl
        uint256 target = maltDataLab.priceTarget();

        uint256 unity = 10**collateralToken.decimals();

        // Convert all balances to be denominated in units of Malt t
        uint256 overflowBalance = maltDataLab.rewardToMaltDecimals((
          address(overflowPool)
        ) * unity) / target);
        uint256 liquidityExtensionBalance = (collateralToken.balance
          address(liquidityExtension)
        ) * unity) / target;
        (
          uint256 swingTraderMaltBalance,
          uint256 swingTraderBalance
        ) = swingTraderManager.getTokenBalances();
        swingTraderBalance = (swingTraderBalance * unity) / target;
```

Since StabilizerNode.stabilize will use the results of maltDataLab.getActualPriceTarget/getSwingTraderEntryPrice to stabilize, and maltDataLab.getActualPriceTarget/getSwingTraderEntryPrice will use `GlobalImpliedCollateralService.collateralRatio` , to ensure correct stabilization, the data in GlobalServiceImpliedCollateralService should be the latest.

```
      function getActualPriceTarget() external view returns (uint256
        uint256 unity = 10**collateralToken.decimals();
```

```
    uint256 icTotal = maltToRewardDecimals(globalIC.collateralRa
...
  function getSwingTraderEntryPrice()
    external
    view
    returns (uint256 stEntryPrice)
  {
    uint256 unity = 10**collateralToken.decimals();
    uint256 icTotal = maltToRewardDecimals(globalIC.collateralRa
```

But since `impliedCollateralService.syncGlobalCollateral()` is not called before StabilizerNode.stabilize calls maltDataLab.getActualPriceTarget/getSwingTraderEntryPrice, this will cause StabilizerNode.stabilize to use stale GlobalImpliedCollateralService data, which will make stabilize incorrect.

A simple example would be:

1. `impliedCollateralService.syncGlobalCollateral()` is called to synchronize the latest data
2. SwingTraderManager.delegateCapital is called, and the collateralToken is taken out from SwingTrader, which will make the `GlobalImpliedCollateralService.collateralRatio` larger than the actual collateralRatio.

```
  function delegateCapital(uint256 amount, address destination)
    external
    onlyRoleMalt(CAPITAL_DELEGATE_ROLE, "Must have capital deleg
    onlyActive
  {
    collateralToken.safeTransfer(destination, amount);
    emit Delegation(amount, destination, msg.sender);
  }
...
  function collateralRatio() public view returns (uint256) {
    uint256 decimals = malt.decimals();
    uint256 totalSupply = malt.totalSupply();
    if (totalSupply == 0) {
      return 0;
    }
    return (collateral.total * (10**decimals)) / totalSupply; //
  }
```

3. When StabilizerNode.stabilize is called, it will use the stale collateralRatio for calculation. If the collateralRatio is too large, the results of maltDataLab.getActualPriceTarget/getSwingTraderEntryPrice will be incorrect, thus making stabilize incorrect.

Since stabilize is a core function of the protocol, stabilizing with the wrong data is likely to cause malt to be depegged, so the vulnerability should be High risk.

Proof of Concept

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L161-L237

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/ImpliedCollateralService.sol#L89-L131

Recommended Mitigation Steps

Call `impliedCollateralService.syncGlobalCollateral()` before StabilizerNode.stabilize calls maltDataLab.getActualPriceTarget.

```
  function stabilize() external nonReentrant onlyEOA onlyActive
    // Ensure data consistency
    maltDataLab.trackPool();

    // Finalize auction if possible before potentially starting
    auction.checkAuctionFinalization();

+   impliedCollateralService.syncGlobalCollateral();

    require(
      block.timestamp >= stabilizeWindowEnd || _stabilityWindow
      "Can't call stabilize"
    );
    stabilizeWindowEnd = block.timestamp + stabilizeBackoffPeri

    // used in 3 location.
    uint256 exchangeRate = maltDataLab.maltPriceAverage(priceAve
    bool stabilizeToPeg = onlyStabilizeToPeg; // gas

    if (!_shouldAdjustSupply(exchangeRate, stabilizeToPeg)) {
      lastStabilize = block.timestamp;
      impliedCollateralService.syncGlobalCollateral();
      return;
    }

    emit Stabilize(block.timestamp, exchangeRate);

    (uint256 livePrice, ) = dexHandler.maltMarketPrice();

    uint256 priceTarget = maltDataLab.getActualPriceTarget();
```

[0xScotch (Malt) confirmed](#)

Medium Risk Findings (16)

[M-01] `priceTarget` is inconsistent in `StabilizerNode.stabilize`

Submitted by [hansfriese](#)

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/StabilityPod/StabilizerNode.sol#L178-L182

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/StabilityPod/StabilizerNode.sol#L294-L298

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/StabilityPod/StabilizerNode.sol#L188

Impact

`priceTarget` is inconsistent in `StabilizerNode.stabilize` so `stabilize` can do auction instead of selling malt and vice versa.

Proof of Concept

In `StabilizerNode.stabilize` , there is an early check using `_shouldAdjustSupply` function.

```
        if (!_shouldAdjustSupply(exchangeRate, stabilizeToPeg)) {
          lastStabilize = block.timestamp;
          impliedCollateralService.syncGlobalCollateral();
          return;
        }
```

In `_shouldAdjustSupply` , `priceTarget` is calculated by `stabilizeToPeg` and then check if `exchangeRate` is outside of some margin of `priceTarget` .

```
        if (stabilizeToPeg) {
          priceTarget = maltDataLab.priceTarget();
        } else {
          priceTarget = maltDataLab.getActualPriceTarget();
        }
```

But in `stabilize` , `priceTarget` is always actual price target of `maltDataLab` regardless of `stabilizeToPeg` . And it decides selling malt or doing auction by the `priceTarget` . So when `stabilizeToPeg` is true, `priceTarget` (= actual price target) can be different from `maltDataLab.priceTarget()` in most cases, and it can cause wrong decision of selling or starting auction after that.

```
        uint256 priceTarget = maltDataLab.getActualPriceTarget();
```

So when `stabilizeToPeg` is true, `stabilize` can do auction instead of selling malt, or vice versa.

Recommended Mitigation Steps

Use same logic as `_shouldAdjustSupply` for `priceTarget` . `priceTarget` should be `maltDataLab.priceTarget()` in `stabilize` when `stabilizeToPeg` is true.

0xScotch (Malt) confirmed

[M-02] The latest malt price can be less than the actual price target and `StabilizerNode.stabilize` will revert

Submitted by hansfriese*, also found by* minhquanym

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/StabilityPod/StabilizerNode.sol#L188

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/StabilityPod/StabilizerNode.sol#L201-L203

Impact

`StabilizerNode.stabilize` will revert when `latestSample < priceTarget`.

Proof of Concept

In StabilizerNode.stabilize, when `exchangeRate > priceTarget` and `_msgSender` is not an admin and not whitelisted, it asserts `livePrice > minThreshold`.

And `minThreshold` is calculated as follows:

```
        uint256 priceTarget = maltDataLab.getActualPriceTarget();



            uint256 latestSample = maltDataLab.maltPriceAverage(0);
            uint256 minThreshold = latestSample -
              (((latestSample - priceTarget) * sampleSlippageBps) /
```

This code snippet assumes that `latestSample >= priceTarget`. Although `exchangeRate > priceTarget`, `exchangeRate` is the malt average price during `priceAveragePeriod`. But `latestSample` is one of those malt prices. So `latestSample` can be less than `exchangeRate` and `priceTarget`, so `stabilize` will revert in this case.

Recommended Mitigation Steps

Use `minThreshold = latestSample + (((priceTarget - latestSample) * sampleSlippageBps) / 10000)` when `priceTarget > latestSample`.

[0xScotch (Malt) confirmed and commented]:

> We actually do want the tx to revert when `latestSample < priceTarget` as that means the most recent sample in the price average feed is below peg but we are in the above peg stabilization flow in the code. However, we do not want the revert to be subtraction overflow as that looks like something went wrong. So we should handle with an explicit error.

[M-03] `LinearDistributor.declareReward` can revert due to dependency of balance

Submitted by hansfriese

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/RewardSystem/LinearDistributor.sol#L147-L151

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/RewardSystem/LinearDistributor.sol#L185-L186

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/RewardSystem/LinearDistributor.sol#L123-L136

Impact

`LinearDistributor.declareReward` will revert and it can cause permanent DOS.

Proof of Concept

In `LinearDistributor.declareReward`, if the balance is greater than the bufferRequirement, the rest will be forfeited.

```
    if (balance > bufferRequirement) {
      // We have more than the buffer required. Forfeit the rest
      uint256 net = balance - bufferRequirement;
      _forfeit(net);
    }
```

And in `_forfeit`, it requires forfeited (= balance - bufferRequirement) <= declaredBalance.

```
  function _forfeit(uint256 forfeited) internal {
    require(forfeited <= declaredBalance, "Cannot forfeit more t
```

So when an attacker sends some collateral tokens to `LinearDistributor`, the balance will be increased and it can cause revert in `_forfeit` and `declareReward`.

Since `declareReward` sends vested amount before `_forfeit` and the vested amount will be increased by time, so this DOS will be temporary.

```
    uint256 distributed = (linearBondedValue * netVest) / vestin
    uint256 balance = collateralToken.balanceOf(address(this));

    if (distributed > balance) {
      distributed = balance;
    }

    if (distributed > 0) {
      // Send vested amount to liquidity mine
      collateralToken.safeTransfer(address(rewardMine), distribu
      rewardMine.releaseReward(distributed);
    }

    balance = collateralToken.balanceOf(address(this));
```

But if the attacker increases the balance enough to cover all reward amount in vesting, `declareReward` will always revert and it can cause permanent DOS.

`decrementRewards` updates `declaredBalance`, but it only decreases `declaredBalance`, so it can't mitigate the DOS.

Recommended Mitigation Steps

Track collateral token balance and add sweep logic for unused collateral tokens in `LinearDistributor`.

Picodes (judge) decreased severity to Medium and commented:

> As this is a DOS scenario where funds are not at risk and the chances that rewards are lost forever are low, downgrading to Medium.

0xScotch (Malt) confirmed and commented:

> I agree this is a DOS vector but a continued attack would require the attacker to spend more and more capital. Should be fixed but doesn't pose any risk of material loss.

[M-04] `SwingTraderManager.swingTraders()` shoudn't contain duplicate `traderContract`s.

Submitted by hansfriese, also found by minhquanym

If `SwingTraderManager.swingTraders()` contains duplicate `traderContract`s, several functions like `buyMalt()` and `sellMalt()` wouldn't work as expected as they work according to traders' balances.

Proof of Concept

During the swing trader addition, there is no validation that each trader should have a unique `traderContract`.

```
    function addSwingTrader(
      uint256 traderId,
      address _swingTrader, //@audit should be unique
      bool active,
      string calldata name
    ) external onlyRoleMalt(ADMIN_ROLE, "Must have admin privs") {
      SwingTraderData storage trader = swingTraders[traderId];
      require(traderId > 2 && trader.id == 0, "TraderId already us
      require(_swingTrader != address(0), "addr(0)");

      swingTraders[traderId] = SwingTraderData({
        id: traderId,
        index: activeTraders.length,
        traderContract: _swingTrader,
        name: name,
        active: active
      });

      activeTraders.push(traderId);

      emit AddSwingTrader(traderId, name, active, _swingTrader);
    }
```

So the same `traderContract` might have 2 or more `traderId` s.

When we check `buyMalt()` as an example, it distributes the ratio according to the trader balance and it wouldn't work properly if one trader contract is counted twice and receives more shares that it can't manage.

Similarly, other functions wouldn't work as expected and return the wrong result.

Recommended Mitigation Steps

Recommend adding a new mapping like `activeTraderContracts` to check if the contract is added already or not.

Then we can check the trader contract is added only once.

[OxScotch (Malt) confirmed]

[M-05] `StabilizerNode.stabilize()` should update `lastTracking` as well to avoid an unnecessary incentive.

Submitted by [hansfriese]

`StabilizerNode.stabilize()` should update `lastTracking` as well to avoid an unnecessary incentive.

Current logic pays unnecessary incentives to track the pool.

Proof of Concept

`trackPool()` pays an incentive per `trackingBackoff` in order to ensure pool consistency.

```
File: 2023-02-malt\contracts\StabilityPod\StabilizerNode.sol
  function trackPool() external onlyActive {
    require(block.timestamp >= lastTracking + trackingBackc
    bool success = maltDataLab.trackPool();
    require(success, "Too early");
    malt.mint(msg.sender, (trackingIncentive * (10**malt.de
    lastTracking = block.timestamp;
    emit Tracking();
  }
```

And `stabilize()` tracks the pool as well and we don't need to pay an incentive unnecessarily in `trackPool()` if `stabilize()` was called recently.

For that, we can update `lastTracking` in `stabilize()`.

Recommended Mitigation Steps

Recommend updating `lastTracking` in `stabilize()`.

```
  function stabilize() external nonReentrant onlyEOA onlyActive
    // Ensure data consistency
    maltDataLab.trackPool();
    lastTracking = block.timestamp; //++++++++++++++++

    ...
```

OxScotch (Malt) confirmed

[M-06] Average `APR`s might be calculated wrongly after calling `populateFromPreviousThrottle()`.

Submitted by hansfriese

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/RewardSystem/RewardThrottle.sol#L660

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/RewardSystem/RewardThrottle.sol#L139

Impact

Average `APR`s might be calculated wrongly after calling `populateFromPreviousThrottle()` and `targetAPR` might be changed unexpectedly.

Proof of Concept

The epoch state struct contains `cumulativeCashflowApr` element and `cashflowAverageApr` is used to adjust `targetAPR` in `updateDesiredAPR()` function.

And `populateFromPreviousThrottle()` is an admin function to change `activeEpoch` and the relevant epoch state using the previous throttle.

And the `activeEpoch` is likely to be increased inside this function.

```
    function populateFromPreviousThrottle(address previousThrottle
      external
      onlyRoleMalt(ADMIN_ROLE, "Only admin role")
    {
      RewardThrottle previous = RewardThrottle(previousThrottle);
      uint256 _activeEpoch = activeEpoch; // gas

      for (uint256 i = _activeEpoch; i < epoch; ++i) {
        (
          uint256 profit,
          uint256 rewarded,
          uint256 bondedValue,
          uint256 desiredAPR,
          uint256 epochsPerYear,
          uint256 cumulativeCashflowApr,
          uint256 cumulativeApr
        ) = previous.epochData(i);

        state[i].bondedValue = bondedValue;
        state[i].profit = profit;
        state[i].rewarded = rewarded;
        state[i].epochsPerYear = epochsPerYear;
        state[i].desiredAPR = desiredAPR;
        state[i].cumulativeCashflowApr = cumulativeCashflowApr;
        state[i].cumulativeApr = cumulativeApr;
      }

      activeEpoch = epoch;
    }
```

The problem might occur when `epoch < _activeEpoch + smoothingPeriod` because `state[epoch].cumulativeCashflowApr` and `state[epoch - smoothingPeriod].cumulativeCashflowApr` will be used for `cashflowAverageApr` calculation.

So `cumulativeCashflowApr` of the original epoch and the newly added epoch will be used together and `cashflowAverageApr` might be calculated wrongly.

As a result, `targetAPR` might be changed unexpectedly.

Recommended Mitigation Steps

Recommend checking `epoch - _activeEpoch > smoothingPeriod` in `populateFromPreviousThrottle()`.

0xScotch (Malt) confirmed

[M-07] `RewardThrottle._sendToDistributor()` reverts if one distributor is inactive.

Submitted by hansfriese

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/RewardSystem/RewardThrottle.sol#L602

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/RewardSystem/LinearDistributor.sol#L101

Impact

`RewardThrottle._sendToDistributor()` reverts if one distributor is inactive.

Proof of Concept

`RewardThrottle._sendToDistributor()` distributes the rewards to several distributors according to their allocation ratios.

```
File: 2023-02-malt\contracts\RewardSystem\RewardThrottle.sol
  function _sendToDistributor(uint256 amount, uint256 epoch
    if (amount == 0) {
      return;
    }

    (
      uint256[] memory poolIds,
      uint256[] memory allocations,
      address[] memory distributors
    ) = bonding.poolAllocations();

    uint256 length = poolIds.length;ratio
    uint256 balance = collateralToken.balanceOf(address(thi
    uint256 rewarded;

    for (uint256 i; i < length; ++i) {
      uint256 share = (amount * allocations[i]) / 1e18;

      if (share == 0) {
        continue;
      }

      if (share > balance) {
        share = balance;
      }

      collateralToken.safeTransfer(distributors[i], share);
```

```
        IDistributor(distributors[i]).declareReward(share); /
```

And `LinearDistributor.declareReward()` has an `onlyActive` modifier and it will revert in case of `inactive`.

```
File: 2023-02-malt\contracts\RewardSystem\LinearDistributor.sol
    function declareReward(uint256 amount)
      external
      onlyRoleMalt(REWARDER_ROLE, "Only rewarder role")
      onlyActive
    {
```

As a result, `RewardThrottle._sendToDistributor()` will revert if one distributor is inactive rather than working with active distributors only.

Recommended Mitigation Steps

I think it's logical to continue to work with active distributors in `_sendToDistributor()`.

[0xScotch (Malt) confirmed](#)

[M-08] `LinearDistributor.declareReward()` might revert after changing `vestingDistributor`.

Submitted by [hansfriese](#)

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/RewardSystem/LinearDistributor.sol#L114

https://github.com/code-423n4/2023-02-malt/blob/main/contracts/RewardSystem/LinearDistributor.sol#L227

Impact

`LinearDistributor.declareReward()` might revert after changing `vestingDistributor` due to uint underflow.

Proof of Concept

In `LinearDistributor.sol`, there is a [setVestingDistributor()](#) function to update `vestingDistributor`.

And in `declareReward()`, it calculates the `netVest` and `netTime` by subtracting the previous amount and time.

```
File: 2023-02-malt\contracts\RewardSystem\LinearDistributor.sol
112:     uint256 currentlyVested = vestingDistributor.getCurrent
113:
114:     uint256 netVest = currentlyVested - previouslyVested; /
115:     uint256 netTime = block.timestamp - previouslyVestedTim
116:
```

But there is no guarantee that the vested amount of the new `vestingDistributor` is greater than the previously saved amount after changing the distributor.

Furthermore, there is no option to change `previouslyVested` beside this declareReward() function and it will keep reverting unless the admin change back the distributor.

Recommended Mitigation Steps

I think it would resolve the above problem if we change the previous amounts as well while updating the distributor.

```
function setVestingDistributor(address _vestingDistributor, ui
  external
  onlyRoleMalt(ADMIN_ROLE, "Must have admin privs")
{
  require(_vestingDistributor != address(0), "SetVestDist: No
  vestingDistributor = IVestingDistributor(_vestingDistributor

  previouslyVested = _previouslyVested;
  previouslyVestedTimestamp = _previouslyVestedTimestamp;
```

```
    }
```

OxScotch (Malt) confirmed and commented:

> Setting `previouslyVested` during the `setVestingDistributor` call seems like a sufficient solution to this.

[M-09] `Repository._removeContract()` removes the contract wrongly.

Submitted by hansfriese, also found by KingNFT

After removing the contract, the `contracts` array would contain the wrong contract names.

Proof of Concept

`Repository._removeContract()` removes the contract name from `contracts` array.

```
File: 2023-02-malt\contracts\Repository.sol
223:   function _removeContract(string memory _name) internal {
224:     bytes32 hashedName = keccak256(abi.encodePacked(_name))
225:     Contract storage currentContract = globalContracts[hash
226:     currentContract.contractAddress = address(0);
227:     currentContract.index = 0;
228:
229:     uint256 index = currentContract.index; //@audit wrong i
230:     string memory lastContract = contracts[contracts.length
231:     contracts[index] = lastContract;
232:     contracts.pop();
233:     emit RemoveContract(hashedName);
234:   }
```

But it uses the already changed index(= 0) and replaces the last name with 0 index all the time.

As a result, the contracts array will still contain the removed name and remove the valid name at index 0.

Recommended Mitigation Steps

We should use the original index like below.

```
function _removeContract(string memory _name) internal {
  bytes32 hashedName = keccak256(abi.encodePacked(_name));
  Contract storage currentContract = globalContracts[hashedNam

  uint256 index = currentContract.index; //++++++++++++++

  currentContract.contractAddress = address(0);
  currentContract.index = 0;

  string memory lastContract = contracts[contracts.length - 1]
  contracts[index] = lastContract;
  contracts.pop();
  emit RemoveContract(hashedName);
}
```

0xScotch (Malt) confirmed

[M-10] StabilizerNode.stabilize may use undistributed rewards in the overflowPool as collateral

Submitted by cccz

In StabilizerNode.stabilize, `globalIC.collateralRatio()` is used to calculate SwingTraderEntryPrice and ActualPriceTarget, with collateralRatio indicating the ratio of the current global collateral to the malt supply.

```
function collateralRatio() public view returns (uint256) {
  uint256 decimals = malt.decimals();
  uint256 totalSupply = malt.totalSupply();
  if (totalSupply == 0) {
    return 0;
  }
  return (collateral.total * (10**decimals)) / totalSupply;
}
```

Global collateral includes the balance of collateral tokens in the overflowPool:

```
function getCollateralizedMalt() public view returns (PoolColl
  uint256 target = maltDataLab.priceTarget(); // 是否选用  getA

  uint256 unity = 10**collateralToken.decimals();

  // Convert all balances to be denominated in units of Malt t
  uint256 overflowBalance = maltDataLab.rewardToMaltDecimals((
    address(overflowPool)
  ) * unity) / target);
  uint256 liquidityExtensionBalance = (collateralToken.balance
    address(liquidityExtension)
  ) * unity) / target;
  (
    uint256 swingTraderMaltBalance,
    uint256 swingTraderBalance
  ) = swingTraderManager.getTokenBalances();
  swingTraderBalance = (swingTraderBalance * unity) / target;

  return
    PoolCollateral({
      lpPool: address(stakeToken),
      // Note that swingTraderBalance also includes the overfl
      // Therefore the total doesn't need to include overflowB
      total: maltDataLab.rewardToMaltDecimals(
          liquidityExtensionBalance + swingTraderBalance
      ),
```

In StabilizerNode.stabilize, since the undistributed rewards in the overflowPool are not distributed, this can cause the actual collateral ratio to be large and thus affect the stabilize process.

A simple example is:

1. `impliedCollateralService.syncGlobalCollateral()` is called to synchronize the latest data.

2. There are some gap epochs in RewardThrottle and their rewards are not distributed from the overflowPool.
3. When StabilizerNode.stabilize is called, it treats the undistributed rewards in the overflowPool as collateral, thus making `globalIC.collateralRatio()` large, and the results of maltDataLab. getActualPriceTarget/getSwingTraderEntryPrice will be incorrect, thus making stabilize incorrect.

Since stabilize is a core function of the protocol, stabilizing with the wrong data is likely to cause malt to be depegged, so the vulnerability should be High risk.

Proof of Concept

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L161-L176

Recommended Mitigation Steps

Call RewardThrottle.checkRewardUnderflow at the beginning of StabilizerNode.stabilize to distribute the rewards in the overflowPool, then call `impliedCollateralService.syncGlobalCollateral()` to synchronize the latest data.

```
  function stabilize() external nonReentrant onlyEOA onlyActive
    // Ensure data consistency
    maltDataLab.trackPool();

    // Finalize auction if possible before potentially starting
    auction.checkAuctionFinalization();

+  RewardThrottle.checkRewardUnderflow();
+  impliedCollateralService.syncGlobalCollateral();

    require(
      block.timestamp >= stabilizeWindowEnd || _stabilityWindow(
      "Can't call stabilize"
    );
```

[0xScotch (Malt) disagreed with severity and commented]():

> By a strict implementation of the protocol this is a bug as it would result in global collateral being slightly misreported and therefore downstream decisions being made on incorrect data. However, in practice, the chances of a big gap in epochs is very low due to the incentivization to upkeep that as well as the degree to which the global IC would be incorrect would be very small. It seems very unlikely this bug would ever lead to a depeg as stated.

> Let's say 50% of the Malt float is in staked LP and the current APR is 10%. We go for 48 epochs (24 hours) without any call to `checkRewardUnderflow`. This means the global IC will be misreported by 24 hours of APR (10%).

> The current APR is 10% and 50% of float is staked, therefore the yearly rewards represent 5% of the total float. One day worth of that is 5% / 365 = 0.013%.

> Therefore we can say that under the above stated circumstances the global IC would be misquoted by 0.02%. Seems very unlikely that discrepancy would be the cause of a depeg.

Picodes (judge) decreased severity to Medium and commented:

> Downgrading to Medium as it indeed seems that the reporting error would remain low and it is unlikely that this could lead to a depeg.

[M-11] RewardThrottle.setTimekeeper: If changing the timekeeper causes the epoch to change, it will mess up the system

Submitted by cccz

RewardThrottle.setTimekeeper allows POOL*UPDATER*ROLE to update the timekeeper when RewardThrottle is active:

```
    function setTimekeeper(address _timekeeper)
      external
      onlyRoleMalt(POOL_UPDATER_ROLE, "Must have pool updater priv
    {
      require(_timekeeper != address(0), "Not address 0");
      timekeeper = ITimekeeper(_timekeeper);
```

```
        }
```

if newTimekeeper.epoch changes, it will cause the following:

1. The newTimekeeper.epoch increases, and the user can immediately call checkRewardUnderflow to fill the gap epoch, thereby distributing a large amount of rewards.
2. The newTimekeeper.epoch decreases, and the contract will use the state of the previous epoch. Since the state.rewarded has reached the upper limit, this will cause the current epoch to be unable to receive rewards.

Proof of Concept

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/RewardSystem/RewardThrottle.sol#L690-L696

Recommended Mitigation Steps

Consider only allowing setTimekeeper to be called when RewardThrottle is not active.

0xScotch (Malt) confirmed and commented:

> This is a good find and I think we will just remove the `setTimekeeper` methods. There is no reason for the timekeeper to ever be updated at this point given all it does it track epochs.

> Historically this method was there because what we now call the timekeeper was called the `MaltDAO` and was earmarked to be used for many other things other than timekeeping. Eventually we realised the timekeeping should be separated into its own thing. These methods were clearly forgotten about and not removed.

[M-12] Value of `totalProfit` might be wrong because of wrong logic in function `sellMalt()`

Submitted by minhquanym, also found by cccz and hansfriese

Contract `SwingTraderManager` has a `totalProfit` variable. It keeps track of total profit swing traders made during `sellMalt()` . However, the logic for accounting is wrong so it will not have the correct value. As the results, it can affect other contracts that integrate with `SwingTraderManager` and use this `totalProfit` variable.

Proof of Concept

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/SwingTraderManager.sol#L252-L258

```
    if (amountSold + dustThreshold >= maxAmount) {
      return maxAmount;
    }

    totalProfit += profit;
    // @audit did not update because already return above

    emit SellMalt(amountSold, profit);
```

Function `sellMalt()` has a dust check before returning result. `totalProfit` should be updated before this check as it returns the value immediately without updating `totalProfit` .

Recommended Mitigation Steps

Updating `totalProfit` before the dust check in function `sellMalt()` .

[0xScotch (Malt) confirmed]()

[M-13] Function `stabilize()` might always revert because of overflow since Malt contract use solidity 0.8

Submitted by [minhquanym]()

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L161

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/DataFeed/MaltDataLab.sol#L326

Impact

MaltDataLab fetched `priceCumulative` directly from Uniswap V2 pool to calculate price of Malt token. However, it is noticed that Uniswap V2 pool use Solidity 0.5.16, which does not revert when overflow happen. In addition, it is actually commented in Uniswap code that

- never overflows, and + overflow is desired

https://github.com/Uniswap/v2-core/blob/ee547b17853e71ed4e0101ccfd52e70d5acded58/contracts/UniswapV2Pair.sol#L77-L81

```
    if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
        // * never overflows, and + overflow is desired
        price0CumulativeLast += uint(UQ112x112.encode(_reserve1).uqc
        price1CumulativeLast += uint(UQ112x112.encode(_reserve0).uqc
    }
```

However, MaltDataLab contracts use Solidity 0.8 and will revert when overflow. It will break the `stabilize()` function and always revert since `stabilize()` call to MaltDataLab contract to get state.

Please note that, with Solidity 0.5.16, when result of addition bigger than `max(uint256)`, it will overflow without any errors. For example, `max(uint256) + 2 = 1`.

So when `price0CumulativeLast` is overflow, the new value of `price0CumulativeLast` will be smaller than old value. As the result, when MaltDataLab doing a subtraction to calculate current price, it might get revert.

Proof of Concept

Function `stabilize()` will call to `MaltDataLab.trackPool()` first:

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L163

```
function stabilize() external nonReentrant onlyEOA onlyActive wh
    // Ensure data consistency
    maltDataLab.trackPool();
    ...
}
```

Function `trackPool()` used a formula that will revert when `priceCumulative` overflow in Uniswap pool.

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/DataFeed/MaltDataLab.sol#L323-L329

```
price = FixedPoint
    .uq112x112(
      uint224(
        // @audit might overflow with solidity 0.8.0
        (priceCumulative - maltPriceCumulativeLast) /
          (blockTimestampLast - maltPriceTimestampLast)
      )
    )
```

Scenario:

1. `maltPriceCumulativeLast = max(uint256 - 10)` and `price = 10`, `timeElapsed = 10`. So the new `priceCumulative = max(uint256 - 10) + 10 * 10 = 99 (overflow)`
2. When doing calculation in Malt protocol, `priceCumulative < maltPriceCumulativeLast`, so `priceCumulative -`

`maltPriceCumulativeLast` will revert and fail

Recommended Mitigation Steps

Consider using `unchecked` block to match handle overflow calculation in Uniswap V2.

[0xScotch (Malt) confirmed]

[M-14] RewardThrottle.populateFromPreviousThrottle may be exposed to front-run attack

Submitted by [cccz]

RewardThrottle.populateFromPreviousThrottle allows ADMIN_ROLE to use epochData from previousThrottle to populate state from activeEpoch to epoch in current RewardThrottle.

```
function populateFromPreviousThrottle(address previousThrottle
  external
  onlyRoleMalt(ADMIN_ROLE, "Only admin role")
{
  RewardThrottle previous = RewardThrottle(previousThrottle);
  uint256 _activeEpoch = activeEpoch; // gas

  for (uint256 i = _activeEpoch; i < epoch; ++i) {
    (
      uint256 profit,
      uint256 rewarded,
      uint256 bondedValue,
      uint256 desiredAPR,
      uint256 epochsPerYear,
      uint256 cumulativeCashflowApr,
      uint256 cumulativeApr
    ) = previous.epochData(i);

    state[i].bondedValue = bondedValue;
    state[i].profit = profit;
    state[i].rewarded = rewarded;
    state[i].epochsPerYear = epochsPerYear;
    state[i].desiredAPR = desiredAPR;
```

```
      state[i].cumulativeCashflowApr = cumulativeCashflowApr;
      state[i].cumulativeApr = cumulativeApr;
    }

    activeEpoch = epoch;
  }
```

But since populateFromPreviousThrottle and `\_fillInEpochGaps` have basically the same function, a malicious user can call fillInEpochGaps to front-run populateFromPreviousThrottle.

```
  function _fillInEpochGaps(uint256 epoch) internal {
    uint256 epochsPerYear = timekeeper.epochsPerYear();
    uint256 _activeEpoch = activeEpoch; // gas

    state[_activeEpoch].bondedValue = bonding.averageBondedValue
    state[_activeEpoch].epochsPerYear = epochsPerYear;
    state[_activeEpoch].desiredAPR = targetAPR;

    if (_activeEpoch > 0) {
      state[_activeEpoch].cumulativeCashflowApr =
        state[_activeEpoch - 1].cumulativeCashflowApr +
        epochCashflowAPR(_activeEpoch - 1);
      state[_activeEpoch].cumulativeApr =
        state[_activeEpoch - 1].cumulativeApr +
        epochAPR(_activeEpoch - 1);
    }

    // Avoid issues if gap between rewards is greater than one e
    for (uint256 i = _activeEpoch + 1; i <= epoch; ++i) {
      if (!state[i].active) {
        state[i].bondedValue = bonding.averageBondedValue(i);
        state[i].profit = 0;
        state[i].rewarded = 0;
        state[i].epochsPerYear = epochsPerYear;
        state[i].desiredAPR = targetAPR;
        state[i].cumulativeCashflowApr =
          state[i - 1].cumulativeCashflowApr +
          epochCashflowAPR(i - 1);
        state[i].cumulativeApr = state[i - 1].cumulativeApr + ep
        state[i].active = true;
      }
    }
```

```
        activeEpoch = epoch;
      }
```

The only difference is that it seems that populateFromPreviousThrottle can make epoch and activeEpoch greater than `timekeeper.epoch()` , thereby updating the state for future epochs, but `\_fillInEpochGaps` makes `activeEpoch = timekeeper.epoch()` , thereby invalidating populateFromPreviousThrottle for future updates. (This usage should be very unlikely).

Proof of Concept

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/RewardSystem/RewardThrottle.sol#L660-L688

Recommended Mitigation Steps

If populateFromPreviousThrottle is used to initialize the state in the current RewardThrottle, it should be called on contract setup.

0xScotch (Malt) confirmed and commented:

> As per #20, we will be removing the `fillInEpochGaps` method.

[M-15] LinearDistributor.declareReward: previouslyVested may update incorrectly, which will cause some rewards to be lost

Submitted by cccz

In LinearDistributor.declareReward, distributed represents the reward to distribute and is calculated using netVest(currentlyVested - previouslyVested).

At the same time, distributed cannot exceed balance, which means that `if balance < linearBondedValue /ast netVest / vestingBondedValue` , part of the rewards in netVest will be lost.

```
        uint256 netVest = currentlyVested - previouslyVested;
        uint256 netTime = block.timestamp - previouslyVestedTimestan

        if (netVest == 0 || vestingBondedValue == 0) {
          return;
        }

        uint256 linearBondedValue = rewardMine.valueOfBonded();

        uint256 distributed = (linearBondedValue * netVest) / vestir
        uint256 balance = collateralToken.balanceOf(address(this));

        if (distributed > balance) {
          distributed = balance;
        }
```

At the end of the function, previouslyVested is directly assigned to currentlyVested instead of using the Vested adjusted according to distributed, which means that the previously lost rewards will also be skipped in the next distribution.

```
        previouslyVested = currentlyVested;
        previouslyVestedTimestamp = block.timestamp;
```

Also, in the next distribution, bufferRequirement will be small because distributed is small, so it may increase the number of forfeits.

```
        if (netTime < buf) {
          bufferRequirement = (distributed * buf * 10000) / netTime
        } else {
          bufferRequirement = distributed;
        }

        if (balance > bufferRequirement) {
          // We have more than the buffer required. Forfeit the rest
          uint256 net = balance - bufferRequirement;
          _forfeit(net);
        }
```

Proof of Concept

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/RewardSystem/LinearDistributor.sol#L111-L153

Recommended Mitigation Steps

Consider adapting previouslyVested based on distributed:

```
        uint256 linearBondedValue = rewardMine.valueOfBonded();

        uint256 distributed = (linearBondedValue * netVest) / vestir
        uint256 balance = collateralToken.balanceOf(address(this));

        if (distributed > balance) {
          distributed = balance;
    +    currentlyVested = distributed * vestingBondedValue / linear
        }
```

0xScotch (Malt) confirmed and commented:

> Finding is correct as stated. I'm not sure how we would ever get into the state required to manifest the bug. Obviously the implementation is incorrect though, so will be fixed.

[M-16] MaltRepository._revokeRole may not work correctly

Submitted by cccz

MaltRepository inherits from AccessControl and adds validation of validRoles to the hasRole function, which means that even if super.hasRole(role, account) == true, if validRoles[role] == false hasRole will return false, which will cause `\_revokeRole` to not work correctly.

```
      function hasRole(bytes32 role, address account)
        public
        view
        override
        returns (bool)
```

```
{
  // Timelock has all possible permissions
  return
    (super.hasRole(role, account) && validRoles[role]) ||
    super.hasRole(TIMELOCK_ROLE, account);
}
```

Consider the case where Alice is granted ADMIN*ROLE, then ADMIN*ROLE is removed in the removeRole function, validRoles[ADMIN_ROLE] == false.

```
function removeRole(bytes32 role) external onlyRole(getRoleAdm
  validRoles[role] = false;
  emit RemoveRole(role);
}
```

Now if the revokeRole function is called on Alice, in the `\_revokeRole`, since hasRole returns false, Alice's ADMIN_ROLE will not be revoked.

Since removeRole ends silently, this may actually cause the caller to incorrectly assume that Alice's ADMIN_ROLE has been revoked:

```
  function _revokeRole(bytes32 role, address account) internal
      if (hasRole(role, account)) {
          _roles[role].members[account] = false;
          emit RoleRevoked(role, account, _msgSender());
      }
  }
```

In addition, the renounceRole and `\_transferRole` functions will also be affected.

In particular, the `\_transferRole` function, if you want to transfer Alice's role to Bob, both Alice and Bob will have the role if validRoles[role]==false.

```
function _transferRole(
  address newAccount,
  address oldAccount,
  bytes32 role
) internal {
    _revokeRole(role, oldAccount);
    _grantRole(role, newAccount);
  }
...
    function renounceRole(bytes32 role, address account) public
        require(account == _msgSender(), "AccessControl: can onl

        _revokeRole(role, account);
    }
```

Proof of Concept

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/Repository.sol#L64-L74

https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/Repository.sol#L99-L102

Recommended Mitigation Steps

Override renounceRole and removeRole in the MaltRepository and modify them as follows:

```

    function renounceRole(bytes32 role, address account) public
+     require(validRoles[role], "Unknown role");
        require(account == _msgSender(), "AccessControl: can onl

        _revokeRole(role, account);
    }
...
    function revokeRole(bytes32 role, address account) public vi
+     require(validRoles[role], "Unknown role");
        _revokeRole(role, account);
    }
...
  function _transferRole(
    address newAccount,
    address oldAccount,
    bytes32 role
  ) internal {
```

```
+  require(validRoles[role], "Unknown role");
     _revokeRole(role, oldAccount);
     _grantRole(role, newAccount);
   }
```

[0xScotch (Malt) confirmed](#)

Low Risk and Non-Critical Issues

For this contest, 4 reports were submitted by wardens detailing low risk and non-critical issues. The [report highlighted below](#) by **hansfriese** received the top score from the judge.

The following wardens also submitted reports: [minhquanym](#), [cccz](#), *and* [KingNFT](#) .

[L-01] `runwayDays` might be longer than it should be due to possible rounding issue

- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/RewardSystem/RewardThrottle.sol#L399
- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/RewardSystem/RewardThrottle.sol#L406

```
    uint256 epochsPerDay = 86400 / timekeeper.epochLength();
    ...
    runwayDays = runwayEpochs / epochsPerDay;
```

When 86400 is not a multiple of `timekeeper.epochLength()` , `runwayDays` might be longer than it should be. Let us assume that `timekeeper.epochLength()` = 43201 (about half a day), and `runwayEpochs` = 360 (about 180 days).

`runwayDays` should be `runwayEpochs * timekeeper.epochLength() / 86400` = 180, but in the above implementation, `epochsPerDay` = 1 and `runwayDays` = 360.

It is recommended to use `runwayDays = runwayEpochs * timekeeper.epochLength() / 86400` directly without the middle variable `epochsPerDay`.

[L-02] `primedBlock` is reset to 0 instead of block.number

- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L224
- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L321-L326

`primedBlock` is reset to 0 instead of `block.number` in `StabilizerNode.stabilize`.

```
      primedBlock = 0;
```

If `primedBlock` = 0, `block.number > primedBlock + primedWindow` holds in most cases and the next caller of `_validateSwingTraderTrigger` will always get default incentive. But this incentive is meaningless.

```
        if (block.number > primedBlock + primedWindow) {
          primedBlock = block.number;
          malt.mint(msg.sender, defaultIncentive * (10**malt.decin
          emit MintMalt(defaultIncentive * (10**malt.decimals()));
          return false;
        }
```

So it is recommended to reset `primedBlock` to `block.number` instead of 0.

[L-03] `skipAuctionThreshold < preferAuctionThreshold` should be checked

- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/Stability

Pod/StabilizerNode.sol#L359-L370

- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L437-L439

```
    if (purchaseAmount > preferAuctionThreshold) {
        ...
    } else {
      _startAuction(originalPriceTarget);
    }
```

```
    if (purchaseAmount < skipAuctionThreshold) {
      return;
    }
```

`skipAuctionThreshold` should be less than `preferAuctionThreshold`.

In `StabilizerNode._triggerSwingTrader`, it starts auction when purchaseAmount <= preferAuctionThreshold.

If `skipAuctionThreshold >= preferAuctionThreshold`, `purchaseAmount <= skipAuctionThreshold` always holds.

So in `_startAuction`, it will never starts an auction and does nothing. So the `stabilize` will not work in this case. It is recommended to check if `skipAuctionThreshold < preferAuctionThreshold` when `skipAuctionThreshold` and `preferAuctionThreshold` are set by the admin.

[L-04] tradeSize will be only 100%, 50%, 33%, ... because of expansionDampingFactor

- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L393-L394

```
uint256 tradeSize = dexHandler.calculateMintingTradeSize(pri
  expansionDampingFactor;
```

`tradeSize` will be only 100%, 50%, 33%, ... of minting trade size calculated from `dexHandler`. I think this is intended, but it can be generalized by basis points or 10**18 so it can support other percentages as follows.

```
uint256 tradeSize = dexHandler.calculateMintingTradeSize(pri
```

[L-05] `updateDesiredAPR` might revert when `aprFloor < maxAdjustment`, so aprFloor(2%) must be greater than maxAdjustment(0.5%)

- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/RewardSystem/RewardThrottle.sol#L131-L183

```
function updateDesiredAPR() public onlyActive {
  ...
  uint256 newAPR = targetAPR; // gas
  uint256 adjustmentCap = maxAdjustment; // gas

  ...

    if (adjustment > adjustmentCap) {
      adjustment = adjustmentCap;
    }

    newAPR -= adjustment;
  }

  uint256 cap = aprCap; // gas
  uint256 floor = aprFloor; // gas
  if (newAPR > cap) {
    newAPR = cap;
  } else if (newAPR < floor) {
    newAPR = floor;
  }
```

```
          targetAPR = newAPR;
          aprLastUpdated = block.timestamp;
          emit UpdateDesiredAPR(newAPR);
        }
```

If `aprFloor < maxAdjustment`, `newAPR` can be `aprFloor` and `adjustment` can be `maxAdjustment`, so `newAPR -= adjustment` will revert. So it needs to make sure that `aprFloor > maxAdjustment`.

[L-06] All balance wasn't sent, some dust would be remained in `_sendToDistributor`

- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/RewardSystem/RewardThrottle.sol#L124-L128
- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/RewardSystem/RewardThrottle.sol#L591-L601

In `RewardThrottle.handleReward`, `_sendToDistributor` is called for left balance.

```
          if (balance > 0) {
            _sendToDistributor(balance, _activeEpoch);
          }

          emit HandleReward(epoch, balance);
```

But in the implementation of `_sendToDistributor`, balance will be split to distributors.

```
            uint256 share = (amount * allocations[i]) / 1e18;
            ...
            collateralToken.safeTransfer(distributors[i], share);
```

So some dust will remain in `RewardThrottle` and the actual rewarded amount can be slightly less than balance in `handleReward`. And the event amount(=balance) will be slightly larger than actual rewarded amount.

[L-07] `_triggerSwingTrader` doesn't try `dexHandler.buyMalt` after `swingTraderManager.buyMalt`

- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L357-L370

`_triggerSwingTrader` doesn't try `dexHandler.buyMalt` after `swingTraderManager.buyMalt`. If capitalUsed is less than purchaseAmount, we can try `dexHandler.buyMalt` with `purchaseAmount - capitalUsed`. But current implementation doesn't try `dexHandler.buyMalt` and it misses possible stabilization.

```
    uint256 purchaseAmount = dexHandler.calculateBurningTradeSiz

    if (purchaseAmount > preferAuctionThreshold) {
      uint256 capitalUsed = swingTraderManager.buyMalt(purchaseA

      uint256 callerCut = (capitalUsed * callerRewardCutBps) / 1

      if (callerCut != 0) {
        malt.mint(msg.sender, callerCut);
        emit MintMalt(callerCut);
      }
    } else {
      _startAuction(originalPriceTarget);
    }
```

[L-08] `swingTraderManager.getTokenBalances` contains inactive swingTrader's balances

- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/SwingTraderManager.sol#L322-L335 swingTraderManager.getTokenBalances doesn't check if swingTrader is active and adds balances regardless of the active status.

```
function getTokenBalances()
  external
  view
  returns (uint256 maltBalance, uint256 collateralBalance)
{
  uint256[] memory traderIds = activeTraders;
  uint256 length = traderIds.length;

  for (uint256 i; i < length; ++i) {
    SwingTraderData memory trader = swingTraders[activeTraders
    maltBalance += malt.balanceOf(trader.traderContract);
    collateralBalance += collateralToken.balanceOf(trader.trac
  }
}
```

But in buyMalt and sellMalt, they only account for balances of active swing traders. This mismatch might cause wrong calculations where getTokenBalances are used.

[L-09] priceTarget seems to be set to wrong value in _triggerSwingTrader

- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L353-L355
- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L303-L306

In StabilizerNode._triggerSwingTrader, priceTarget is set to icTotal when exchangeRate < icTotal.

```
if (exchangeRate < icTotal) {
  priceTarget = icTotal;
}
```

If `icTotal` is slightly greater than `exchangeRate`, `priceTarget` can be `exchangeRate` + dust.

But in `_shouldAdjustSupply`, `exchangeRate` should be less than some margin of `priceTarget` to proceed actual stabilization.

```
        return
         (exchangeRate <= (priceTarget - lowerThreshold) &&
           !auction.auctionExists(auction.currentAuctionId())) ||
         exchangeRate >= (priceTarget + upperThreshold);
```

So the `priceTarget` updating logic seems incorrect.

[N-01] Typo

- https://github.com/code-423n4/2023-02-malt/blob/700f9b468f9cf8c9c5cffaa1eba1b8dea40503f9/contracts/StabilityPod/StabilizerNode.sol#L411

sandwich is not correct here.

0xScotch (Malt) confirmed

Gas Optimizations

For this contest, 3 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **cccz** received the top score from the judge.

The following wardens also submitted reports: hansfriese and minhquanym.

[G-01] Increments can be unchecked

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

https://github.com/ethereum/solidity/issues/10695

Instances include:

```
MaltRepository.grantRoleMultiple()#132:    for (uint256 i; i < l
MaltRepository._setup()#188:    for (uint256 i; i < length; ++i)
RewardThrottle.checkRewardUnderflow()#446:      for (uint256 i =
RewardThrottle._sendToDistributor()#590:    for (uint256 i; i <
RewardThrottle._fillInEpochGaps()#639:    for (uint256 i = _acti
RewardThrottle.populateFromPreviousThrottle()#667:    for (uint2
SwingTraderManager.buyMalt()#154:    for (uint256 i; i < length;
SwingTraderManager.buyMalt()#170:    for (uint256 i; i < length;
SwingTraderManager.sellMalt()#208:    for (uint256 i; i < lengt
SwingTraderManager.sellMalt()#224:    for (uint256 i; i < lengt
SwingTraderManager.costBasis()#269:    for (uint256 i; i < lengt
SwingTraderManager.calculateSwingTraderMaltRatio()#300:    for (
SwingTraderManager.getTokenBalances()#330:    for (uint256 i; i
SwingTraderManager.delegateCapital()#348:    for (uint256 i; i <
SwingTraderManager.delegateCapital()#366:    for (uint256 i; i <
SwingTraderManager.deployedCapital()#389:    for (uint256 i; i <
```

The code would go from:

```
for (uint256 i; i < numIterations; ++i) {
 // ...
}
```

to

```
for (uint256 i; i < numIterations;) {
 // ...
 unchecked { ++i; }
}
```

[G-02] `GlobalImpliedCollateralService.swingTraderCollateralRatio()` : should use memory instead of storage variable

See @audit tag

```
function swingTraderCollateralRatio() public view returns (uin
  uint256 decimals = malt.decimals();
  uint256 totalSupply = malt.totalSupply();

  if (totalSupply == 0) {
    return 0;
  }

  return (collateral.swingTrader * (10**decimals)) / malt.tota
}
```

[G-03] `SwingTraderManager.buyMalt()` : should use memory instead of storage variable

See @audit tag

```
    uint256[] memory traderIds = activeTraders;
    uint256 length = traderIds.length;

    uint256 totalCapital;
    uint256[] memory traderCapital = new uint256[](length);

    for (uint256 i; i < length; ++i) {
      SwingTraderData memory trader = swingTraders[activeTraders

      if (!trader.active) {
        continue;
      }

      uint256 traderBalance = collateralToken.balanceOf(trader.t
      totalCapital += traderBalance;
      traderCapital[i] = traderBalance;
    }

    if (totalCapital == 0) {
      return 0;
    }

    for (uint256 i; i < length; ++i) {
      SwingTraderData memory trader = swingTraders[activeTraders
      uint256 share = (maxCapital * traderCapital[i]) / totalCap
```

```
  if (share == 0) {
    continue;
  }
```

[G-04] `SwingTraderManager.sellMalt()` : should use memory instead of storage variable

See @audit tag

```
uint256[] memory traderIds = activeTraders;
uint256 length = traderIds.length;
uint256 profit;

uint256 totalMalt;
uint256[] memory traderMalt = new uint256[](length);

for (uint256 i; i < length; ++i) {
  SwingTraderData memory trader = swingTraders[activeTraders

  if (!trader.active) {
    continue;
  }

  uint256 traderMaltBalance = malt.balanceOf(trader.traderC
  totalMalt += traderMaltBalance;
  traderMalt[i] = traderMaltBalance;
}

if (totalMalt == 0) {
  return 0;
}

for (uint256 i; i < length; ++i) {
  SwingTraderData memory trader = swingTraders[activeTraders
```

[G-05] `SwingTraderManager.costBasis()` : should use memory instead of storage variable

See @audit tag

```
    uint256[] memory traderIds = activeTraders;
    uint256 length = traderIds.length;
    decimals = collateralToken.decimals();

    uint256 totalMaltBalance;
    uint256 totalDeployedCapital;

    for (uint256 i; i < length; ++i) {
      SwingTraderData memory trader = swingTraders[activeTraders
```

[G-06] `SwingTraderManager.calculateSwingTraderMaltRatio()` : should use memory instead of storage variable

See @audit tag

```
  function calculateSwingTraderMaltRatio()
    public
    view
    returns (uint256 maltRatio)
  {
    uint256[] memory traderIds = activeTraders;
    uint256 length = traderIds.length;
    uint256 decimals = collateralToken.decimals();
    uint256 maltDecimals = malt.decimals();
    uint256 totalMaltBalance;
    uint256 totalCollateralBalance;

    for (uint256 i; i < length; ++i) {
      SwingTraderData memory trader = swingTraders[activeTraders
```

[G-07] `SwingTraderManager.getTokenBalances()` : should use memory instead of storage variable

See @audit tag

```
  function getTokenBalances()
    external
    view
    returns (uint256 maltBalance, uint256 collateralBalance)
  {
    uint256[] memory traderIds = activeTraders;
    uint256 length = traderIds.length;

    for (uint256 i; i < length; ++i) {
      SwingTraderData memory trader = swingTraders[activeTraders
```

[G-08] `SwingTraderManager.delegateCapital()`: should use memory instead of storage variable

See @audit tag

```
function delegateCapital(uint256 amount, address destination)
  external
  onlyRoleMalt(CAPITAL_DELEGATE_ROLE, "Must have capital deleg
  onlyActive
{
  uint256[] memory traderIds = activeTraders;
  uint256 length = traderIds.length;

  uint256 totalCapital;
  uint256[] memory traderCapital = new uint256[](length);

  for (uint256 i; i < length; ++i) {
    SwingTraderData memory trader = swingTraders[activeTraders

    if (!trader.active) {
      continue;
    }

    uint256 traderBalance = collateralToken.balanceOf(trader.t
    totalCapital += traderBalance;
    traderCapital[i] = traderBalance;
  }

  if (totalCapital == 0) {
    return;
  }

  uint256 capitalUsed;

  for (uint256 i; i < length; ++i) {
      SwingTraderData memory trader = swingTraders[activeTraders
```

[G-09] `SwingTraderManager.deployedCapital()`: should use memory instead of storage variable

See @audit tag

```
  function deployedCapital() external view returns (uint256 depl
    uint256[] memory traderIds = activeTraders;
    uint256 length = traderIds.length;

    for (uint256 i; i < length; ++i) {
      SwingTraderData memory trader = swingTraders[activeTraders
```

[G-10] `GlobalImpliedCollateralService.sync()`: existingPool.* should get cached

See @audit tag

```
    uint256 existingCollateral = existingPool.total;

    uint256 total = collateral.total; // gas
    if (existingCollateral <= total) {
      total -= existingCollateral; // subtract existing value
    } else {
      total = 0;
    }

    uint256 swingTraderMalt = collateral.swingTraderMalt; // gas
    if (existingPool.swingTraderMalt <= swingTraderMalt) {
      swingTraderMalt -= existingPool.swingTraderMalt;
    } else {
      swingTraderMalt = 0;
    }

    uint256 swingTraderCollat = collateral.swingTrader; // gas
    if (existingPool.swingTrader <= swingTraderCollat) {
      swingTraderCollat -= existingPool.swingTrader;
    } else {
      swingTraderCollat = 0;
```

```
  }

  uint256 arb = collateral.arbTokens; // gas
  if (existingPool.arbTokens <= arb) {
    arb -= existingPool.arbTokens;
  } else {
    arb = 0;
  }

  uint256 overflow = collateral.rewardOverflow; // gas
  if (existingPool.rewardOverflow <= overflow) {
    overflow -= existingPool.rewardOverflow;
  } else {
    overflow = 0;
  }

  uint256 liquidityExtension = collateral.liquidityExtension;
  if (existingPool.liquidityExtension <= liquidityExtension) {
    liquidityExtension -= existingPool.liquidityExtension;
  } else {
    liquidityExtension = 0;
  }
```

[G-11] `LinearDistributor.decrementRewards()`: declaredBalance should get cached

See @audit tag

```
function decrementRewards(uint256 amount)
  external
  onlyRoleMalt(REWARD_MINE_ROLE, "Only reward mine")
{
  require(
    amount <= declaredBalance,                          //@audi
    "Can't decrement more than total reward balance"
  );

  if (amount > 0) {
    declaredBalance = declaredBalance - amount;         //@audi
  }
}
```

[G-12] `LinearDistributor._forfeit()` : declaredBalance should get cached

See @audit tag

```
    function _forfeit(uint256 forfeited) internal {
      require(forfeited <= declaredBalance, "Cannot forfeit more t

      declaredBalance = declaredBalance - forfeited;
```

[G-13] `SwingTraderManager.buyMalt()` : swingTraders should get cached

See @audit tag

```
      for (uint256 i; i < length; ++i) {
        SwingTraderData memory trader = swingTraders[activeTraders

        if (!trader.active) {
          continue;
        }

        uint256 traderBalance = collateralToken.balanceOf(trader.t
        totalCapital += traderBalance;
        traderCapital[i] = traderBalance;
      }

      if (totalCapital == 0) {
        return 0;
      }

      for (uint256 i; i < length; ++i) {
        SwingTraderData memory trader = swingTraders[activeTraders
```

[G-14] `SwingTraderManager.sellMalt()` : swingTraders should get cached

See @audit tag

```
          SwingTraderData memory trader = swingTraders[activeTraders

          if (!trader.active) {
            continue;
          }

          uint256 traderMaltBalance = malt.balanceOf(trader.traderC
          totalMalt += traderMaltBalance;
          traderMalt[i] = traderMaltBalance;
        }

        if (totalMalt == 0) {
          return 0;
        }

        for (uint256 i; i < length; ++i) {
          SwingTraderData memory trader = swingTraders[activeTraders
          uint256 share = (maxAmount * traderMalt[i]) / totalMalt;
```

[G-15] `SwingTraderManager.delegateCapital()` : swingTraders should get cached

See @audit tag

```
        for (uint256 i; i < length; ++i) {
          SwingTraderData memory trader = swingTraders[activeTraders

          if (!trader.active) {
            continue;
          }

          uint256 traderBalance = collateralToken.balanceOf(trader.t
          totalCapital += traderBalance;
          traderCapital[i] = traderBalance;
        }

        if (totalCapital == 0) {
          return;
        }

        uint256 capitalUsed;
```

```
    for (uint256 i; i < length; ++i) {
      SwingTraderData memory trader = swingTraders[activeTraders
      uint256 share = (amount * traderCapital[i]) / totalCapital
```

[G-16] LinearDistributor.sol has code that needs to be UNCHECKED in many places

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an unchecked block: https://docs.soliditylang.org/en/v0.8.7/control-structures.html#checked-or-unchecked-arithmetic

L149 SHOULD BE UNCHECKED DUE TO L147

L169 SHOULD BE UNCHECKED DUE TO L164

L188 SHOULD BE UNCHECKED DUE TO L186

```
147:    if (balance > bufferRequirement) {
148:      // We have more than the buffer required. Forfeit the
149:      uint256 net = balance - bufferRequirement;
150:      _forfeit(net);
151:    }
...
163:    require(
164:      amount <= declaredBalance,
165:      "Can't decrement more than total reward balance"
166:    );
167:
168:    if (amount > 0) {
169:      declaredBalance = declaredBalance - amount;
170:    }
...
185:  function _forfeit(uint256 forfeited) internal {
186:    require(forfeited <= declaredBalance, "Cannot forfeit mo
187:
188:    declaredBalance = declaredBalance - forfeited;
```

[G-17] RewardThrottle.sol has code that needs to be UNCHECKED in many places

L117 SHOULD BE UNCHECKED DUE TO L116

L162 AND L146 SHOULD BE UNCHECKED DUE TO L145

L222 SHOULD BE UNCHECKED DUE TO L219 ...

```
116:         if (balance > remainder) {
117           balance -= remainder;
...
145     if (cashflowAverageApr > targetCashflowApr) {
146       uint256 delta = cashflowAverageApr - targetCashflowApr;
...
162:       uint256 delta = targetCashflowApr - cashflowAverageApr
...
219     if (endEpoch < averagePeriod) {
      averagePeriod = currentEpoch;
    } else {
222:       startEpoch = endEpoch - averagePeriod;
...
```

0xScotch (Malt) confirmed

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top