Malt Finance contest
Findings & Analysis Report

2022-02-14

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 code 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 code contest outlined in this document, C4 conducted an analysis of Malt Finance contest smart contract system written in Solidity. The code contest took place between November 25—December 1 2021.

Wardens

35 Wardens contributed reports to the Malt Finance contest:

1. WatchPug (jtp and ming)
2. gzeon
3. leastwood
4. cmichel
5. harleythedog
6. 0x0x0x
7. jayjonah8
8. Meta0xNull
9. hyh
10. pauliax
11. ScopeLift (wildmolasses, bendi, mds1)
12. defsec
13. gpersoon
14. TomFrench
15. stonesandtrees
16. xYrYuYx
17. GiveMeTestEther
18. 0x1f8b
19. yeOlde
20. nathaniel
21. robee
22. pmerkleplant
23. 0xwags

24. Koustre
25. danb
26. BouSalman
27. sabtikw
28. hagrid
29. tabish
30. loop
31. thank_you
32. shenwilly

This contest was judged by Alex the Entreprenerd.

Final report assembled by CloudEllie and itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 78 unique vulnerabilities and 178 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 5 received a risk rating in the category of HIGH severity, 30 received a risk rating in the category of MEDIUM severity, and 43 received a risk rating in the category of LOW severity.

C4 analysis also identified 25 non-critical recommendations and 75 gas optimizations.

Scope

The code under review can be found within the C4 Malt Finance contest repository, and is composed of 24 smart contracts written in the Solidity programming language and includes 4,452 source lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

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

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

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](https://code4rena.com).

High Risk Findings (5)

[H-01] Timelock can be bypassed

Submitted by WatchPug, also found by 0x0x0x and gzeon

The purpose of a Timelock contract is to put a limit on the privileges of the `governor`, by forcing a two step process with a preset delay time.

However, we found that the current implementation actually won't serve that purpose as it allows the `governor` to execute any transactions without any constraints.

To do that, the current governor can call `Timelock#setGovernor(address _governor)` and set a new `governor` effective immediately.

And the new `governor` can then call `Timelock#setDelay()` and change the delay to `0`, also effective immediately.

The new `governor` can now use all the privileges without a delay, including granting minter role to any address and mint unlimited amount of MALT.

In conclusion, a Timelock contract is supposed to guard the protocol from lost private key or malicious actions. The current implementation won't fulfill that mission.

https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/Timelock.sol#L98-L105

```
function setGovernor(address _governor)
  public
  onlyRole(GOVERNOR_ROLE, "Must have timelock role")
{
  _swapRole(_governor, governor, GOVERNOR_ROLE);
  governor = _governor;
  emit NewGovernor(_governor);
}
```

https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/Timelock.sol#L66-L77

```
function setDelay(uint256 _delay)
  public
  onlyRole(GOVERNOR_ROLE, "Must have timelock role")
{
  require(
    _delay >= 0 && _delay < gracePeriod,
    "Timelock::setDelay: Delay must not be greater equal to ze
  );
  delay = _delay;

  emit NewDelay(delay);
}
```

Recommendation

Consider making `setGovernor` and `setDelay` only callable from the Timelock contract itself.

Specificaly, changing from `onlyRole(GOVERNOR_ROLE, "Must have timelock role")` to `require(msg.sender == address(this), "...")`.

Also, consider changing `_adminSetup(_admin)` in `Timelock#initialize()` to `_adminSetup(address(this))`, so that all roles are managed by the timelock itself as well.

[0xScotch (sponsor) confirmed]

[Alex the Entreprenerd (judge) commented]:

> The warden has identified an exploit that allows to sidestep the delay for the timelock, effectively bypassing all of the timelock's security guarantees. Because of the gravity of this, I agree with the high risk severity.

> Mitigation can be achieved by ensuring that all operations run under a time delay

[H-02] Unable to remove liquidity in Recovery Mode

Submitted by gzeon

According to https://github.com/code-423n4/2021-11-malt#high-level-overview-of-the-malt-protocol

> When the Malt price TWAP drops below a specified threshold (eg 2% below peg) then the protocol will revert any transaction that tries to remove Malt from the AMM pool (ie buying Malt or removing liquidity). Users wanting to remove liquidity can still do so via the UniswapHandler contract that is whitelisted in recovery mode.

However, in https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/DexHandlers/UniswapHandler.sol#L236 liquidity removed is directly sent to msg.sender, which would revert if it is not whitelisted https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/PoolTransferVerification.sol#L53

Recommended Mitigation Steps

Liquidity should be removed to UniswapHandler contract, then the proceed is sent to msg.sender

0xScotch (sponsor) confirmed

Alex the Entreprenerd (judge) commented:

> I believe this finding to be correct, because of the whitelisting on `verifyTransfer`, during recovery mode the removal of liquidity from UniSwapV2Pair will perform safeTransfers: https://github.com/Uniswap/v2-core/blob/4dd59067c76dea4a0e8e4bfdda41877a6b16dedc/contracts/UniswapV2Pair.sol#L148

> This means that the `_beforeTokenTransfer` will be called which eventually will call `verifyTransfer` which, if the price is below peg will revert.

> Transfering the funds to the whitelisted contract should avoid this issue.

> I'd like to remind the sponsor that anyone could deploy similar swapping contracts (or different ones such as curve), so if a person is motivate enough, all the whitelisting could technically be sidestepped.

> That said, given the condition of LPing on Uniswap, the check and the current system would make it impossible to withdraw funds.

> Because this does indeed compromises the availability of funds (effectively requiring the admin to unstock them manually via Whitelisting each user), I agree with High Severity

[H-03] getAuctionCore function returns wrong values out of order

Submitted by jayjonah8

Impact

In the `AuctionEscapeHatch.sol` file both `earlyExitReturn()` and `\_calculateMaltRequiredForExit` call the `getAuctionCore()` function which has 10 possible return values most of which are not used. It gets the wrong value back for the “active” variable since it’s the 10th argument but both functions have it as the 9th return value where “preAuctionReserveRatio” should be because of one missing comma. This is serious because these both are functions which deal with allowing a user to exit their arbitrage token position early. This can result in a loss of user funds.

Proof of Concept

- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/AuctionEscapeHatch.sol#L100
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/AuctionEscapeHatch.sol#L174
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Auction.sol#L527

Tools Used

Manual code review

Recommended Mitigation Steps

In `AuctionEscapeHatch.sol` change the following in `\_calculateMaltRequiredForExit()` and earlyExitReturn() functions:

From:

(,,,,, uint256 pegPrice, , uint256 auctionEndTime, bool active) = auction.getAuctionCore(_auctionId);

To:

(,,,,, uint256 pegPrice, , uint256 auctionEndTime, , bool active) = auction.getAuctionCore(_auctionId);

[0xScotch (sponsor) confirmed]()

[Alex the Entreprenerd (judge) commented]():

> The warden identified a mistake in programming where the code would use the wrong returned value. Because of this, the entire protocol functionality can be compromised. As such I agree with High Severity

[H-04] `AuctionBurnReserveSkew.getPegDeltaFrequency()` Wrong implementation can result in an improper amount of excess Liquidity Extension balance to be used at the end of an auction

Submitted by WatchPug

https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/AuctionBurnReserveSkew.sol#L116-L132

```
  function getPegDeltaFrequency() public view returns (uint256) {
    uint256 initialIndex = 0;
    uint256 index;

    if (count > auctionAverageLookback) {
      initialIndex = count - auctionAverageLookback;
    }

    uint256 total = 0;

    for (uint256 i = initialIndex; i < count; ++i) {
      index = _getIndexOfObservation(i);
      total = total + pegObservations[index];
    }

    return total * 10000 / auctionAverageLookback;
  }
```

When `count < auctionAverageLookback`, at L131, it should be `return total * 10000 / count;`. The current implementation will return a smaller value than expected.

The result of `getPegDeltaFrequency()` will be used for calculating `realBurnBudget` for auctions. With the result of `getPegDeltaFrequency()` being inaccurate, can result in an improper amount of excess Liquidity Extension balance to be used at the end of an auction.

0xScotch (sponsor) confirmed and disagreed with severity:

> I actually think this should be higher severity. This bug could manifest in liquidity extension being depleted to zero which could have catastrophic consequences downstream.

Alex the Entreprenerd (judge) commented:

> Agree with the finding, this is an incorrect logic in the protocol, which can limit it's functionality and as the sponsor says: `could have catastrophic consequences downstream` as such I'll increase the severity to high.

> Mitigation seems to be straightforward

[H-05] AuctionEschapeHatch.sol#exitEarly updates state of the auction wrongly

Submitted by 0x0x0x

`AuctionEschapeHatch.sol#exitEarly` takes as input `amount` to represent how much of the

When the user exits an auction with profit, to apply the profit penalty less `maltQuantity` is liquidated compared to how much malt token the liquidated amount corresponds to. The problem is `auction.amendAccountParticipation()` simply subtracts the malt quantity with penalty and full `amount` from users auction stats. This causes a major problem, since in `_calculateMaltRequiredForExit` those values are used for calculation by calculating maltQuantity as follow:

```
uint256 maltQuantity =
userMaltPurchased.mul(amount).div(userCommitment);
```

The ratio of `userMaltPurchased / userCommitment` gets higher after each profit taking (since penalty is applied to substracted `maltQuantity` from `userMaltPurchased`), by doing so a user can earn more than it should. Since after each profit taking users commitment corresponds to proportionally more malt, the user can even reduce profit penalties by dividing `exitEarly` calls in several calls.

In other words, the ratio of `userMaltPurchased / userCommitment` gets higher after each profit taking and user can claim more malt with less commitment. Furthermore after all `userMaltPurchased` is claimed the user can have `userCommitment` left over, which can be used to `claimArbitrage`, when possible.

Mitigation Step

Make sure which values are used for what and update values which doesn't create problems like this. Rethink about how to track values of an auction correctly.

[0xScotch (sponsor) confirmed]

[Alex the Entreprenerd (judge) commented]:

> The warden has identified an exploit that allows early withdrawers to gain more rewards than expected. Anytime "points" and rewards need to be earned over time, it's ideal to accrue points in order to distribute them (see how Compound or AAVE tokens work) Because the warden showed a flow in the accounting logic for the protocol, I agree with high severity.

Medium Risk Findings (30)

[M-01] TIMELOCK_ROLE Has Absolute Power to Withdraw All FUND May Raise Red Flags for Investors

Submitted by Meta0xNull

`TIMELOCK_ROLE` Can Withdraw All FUND from the Contracts via `emergencyWithdrawGAS()`, `emergencyWithdraw()`, `partialWithdrawGAS()`, `partialWithdraw()`.

While I believe developer have good intention to use these functions. It often associate with Rug Pull by developer in the eyes of investors because Rug Pull is not uncommon in Defi. Investors lose all their hard earn money.

Read More: $10.8M Stolen, Developers Implicated in Alleged Smart Contract 'Rug Pull' https://www.coindesk.com/tech/2020/12/02/108m-stolen-developers-implicated-in-alleged-smart-contract-rug-pull/

Read More: The Rise of Cryptocurrency Exit Scams and DeFi Rug Pulls https://www.cylynx.io/blog/the-rise-of-cryptocurrency-exit-scams-and-defi-rug-pulls/

Proof of Concept

https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Permissions.sol#L80-L109

Recommended Mitigation Steps

1. Pause the Contract and Disable All Functions when Bad Thing Happen rather than Withdraw All Fund to a random address.
2. If Withdraw Fund can't avoid, a Multi Sig ETH Address should be hardcoded into the contract to ensure the fund move to a safe wallet.

[0xScotch (sponsor) commented]():

> Duplicate of #263

[Alex the Entreprenerd (judge) commented]():

> This is not a duplicate of #263, where 263 talks about sidestepping the delay of the timelock, this finding talks about the high degree of power that the TIMELOCK_ROLE has.

> This is a typical "admin privilege" finding, it's very important to disclose admin privileges to users so that they can make informed decisions

> In this case the TIMELOCK_ROLE can effectively rug the protocol, however this is contingent on the account that has the role to pull the rug.

> Because of its reliance on external factors, am downgrading the finding to medium severity

[M-02] Frontrunning in UniswapHandler calls to UniswapV2Router

*Submitted by thank*you, also found by 0x0x0x, cmichel, defsec, harleythedog, hyh, Koustre, leastwood, Meta0xNull, pauliax, pmerkleplant, tabish, WatchPug, and xYrYuYx_

UniswapHandler utilizes UniswapV2Router to swap, add liquidity, and remove liquidity with the UniswapV2Pair contract. In order to utilize these functionalities, UniswapHandler must call various UniswapV2Router methods.

- addLiquidity
- removeLiquidity
- swapExactTokensForTokens (swaps for both DAI and Malt)

In all three methods, UniswapV2Router requires the callee to provide input arguments that define how much the amount out minimum UniswapHandler will allow for a trade. This argument is designed to prevent slippage and more importantly, sandwich attacks.

UniswapHandler correctly handles price slippage when calling addLiquidity. However, that is not the case for removeLiquidity and swapExactTokensForTokens here and here. For both methods, 0 is passed in as the amount out minimum allowed for a trade. This allows for anyone watching the mempool to sandwich attack UniswapHandler (or any contract that calls UniswapHandler) in such a way that allows the hacker to profit off of a guaranteed trade.

How does this work? Let's assume UniswapHandler makes a call to UniswapV2Router#swapExactTokensForTokens to trade DAI for Malt. Any hacker who watches the mempool and sees this transaction can immediately buy as much Malt as they want. This raises the price of Malt. Since UniswapHandler is willing to accept any amount out minimum (the number is set to zero), then the UniswapHandler will always trade DAI for Malt. This second transaction raises the price of Malt even further. Finally, the hacker trades their Malt for DAI, receiving a profit due to the artificially inflated price of Malt from the sandwich attack.

It's important to note that anyone has access to the UniswapV2Router contract. There are no known ACL controls on UniswapV2Router. This sandwich attack can impact even the `buyMalt` function.

The following functions when called are vulnerable to frontrunning attacks:

- UniswapHandler#buyMalt
- UniswapHandler#sellMalt
- UniswapHandler#removeLiquidity

And by extension the following contract functions since they also call the UniswapHandler function calls:

- Bonding#unbondAndBreak
- LiquidityExtension#purchaseAndBurn
- RewardReinvestor#splitReinvest
- StabilizerNode#stabilize
- SwingTrader#buyMalt

Proof of Concept

Refer to the impact section for affected code and links to the appropriate LoC.

Recommended Mitigation Steps

The UniswapV2Router and UniswapV2Pair contract should allow only the UniswapHandler contract to call either contract. In addition, price slippage checks should be implemented whenever removing liquidity or swapping tokens. This ensures that a frontrunning attack can't occur.

Anything Else We Should Know

I wish I had more time to work on this bug but unfortunately I have several current clients who require significant time from me. I'm happy to pursue this beyond the initial submission, in particular building a concrete PoC. I think the most important takeaway from this bug find is that anyone can purchase Malt at any time and anyone can manipulate the Malt reserve. This in turn impacts other functionalities that rely on the Malt reserve to make price/token calculations such as exiting an auction early or reinvesting rewards.

[0xScotch (sponsor) confirmed]

[Alex the Entreprenerd (judge) commented]:

> Because transactions sit on the mempool (which is public, hence accessible by anyone), they can be frontrun, because of this swapping protocols (uniswap in this case) offer slippage checks. Setting the slippage checks allows a frontrunner to squeeze the maximum amount of value possible (sometimes the whole amount).

> Because this applies to a leak of value, I believe medium severity to be correct

[M-03] AbstractRewardMine.sol#setRewardToken is dangerous

Submitted by 0x0x0x, also found by harleythedog and hyh

In case the reward token is changed, `totalDeclaredReward` will be changed and likely equal to `0`. Since `_userStakePadding` and `_globalStakePadding` are accumulated, changing the reward token will not reset those values. Thus, it will create problems.

Recommendation

I think it would be the best to remove this function.

If you want to keep it, then it must have an event and it should be used by a timelock contract. Furthermore, it has to be used carefully and the new token should be distributed such that padding variables still make sense.

[0xScotch (sponsor) confirmed]

[Alex the Entreprenerd (judge) commented]:

> Agree with highlighting the Admin Privilege, however because this is contingent on a malicious Admin, I'll downgrade the finding to Medium Severity.

> Mitigation could be done by ensuring old rewards are sent out, still claimable, or by making the `rewardToken` immutable

[M-04] The Power Structure is Too Centralized And Protocol May Break If Anything Happen to Admin

Submitted by Meta0xNull

There are a lot of different roles in Malt Finance to handle different tasks. All these roles only can set by Admin. If anything happen to Admin and he/she no longer available, the protocol will start countdown to the end of life.

Proof of Concept

- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Auction.sol#L890-L1013
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/RewardSystem/RewardDistributor.sol#L291-L345
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Bonding.sol#L327-L355 Many More Sol...

Recommended Mitigation Steps

1. Some tasks don't really need a special role like StabilizerNode. Should allow any community members to run their own StabilizerNode without approval needed.
2. Consider transfer Admin to Multisig or DAO.

0xScotch (sponsor) disputed:

> This is a known risk that is by design. We will migrate to DAO control of parameters when the protocol matures.

Alex the Entreprenerd (judge) commented:

> I respect the wardens work for flagging this up. Because any of these exploits are contingent on the Admin being malicious, I'll downgrade to Medium Severity

[M-05] _notSameBlock() can be circumvented in bondToAccount()

Submitted by gpersoon, also found by leastwood and ScopeLift

The function bondToAccount() of Bonding.sol has a check based on _notSameBlock() _notSameBlock() makes sure the same msg.sender cannot do 2 actions within the same block.

However this can be circumvented in this case: Suppose you call bondToAccount() via a (custom) smart contract, then the msg.sender will be the address of the smart contract. For a pseudo code proof of concept see below.

I'm not sure what the deeper reason is for the _notSameBlock() in bondToAccount(). But if it is important then circumventing this check it will pose a risk.

Proof of Concept

call function attack1.attack()

```
contract attack1 {
   function attack(address account, uint256 amount) {
         call attack2.forward(account, amount);
         call any other function of malt
  }
}

contract attack2 {
   function forward(address account, uint256 amount) {
       call bonding.bondToAccount(account, amount); // uses msg.
   }
}
```

https://github.com/code-423n4/2021-11-malt/blob/d3f6a57ba6694b47389b16d9d0a36a956c5e6a94/src/contracts/Bonding.sol#L81-L92

```
function bondToAccount(address account, uint256 amount) public {
    if (msg.sender != offering) {
         _notSameBlock();
```

```
        }
        ...
```

https://github.com/code-423n4/2021-11-malt/blob/d3f6a57ba6694b47389b16d9d0a36a956c5e6a94/src/contracts/Permissions.sol#L135-L141

```
    function _notSameBlock() internal {
        require( block.number > lastBlock[_msgSender()],"Can't carry
        lastBlock[_msgSender()] = block.number;
      }
```

Recommended Mitigation Steps

Add access controls to the function bondToAccount() An end-user could still call bond()

0xScotch (sponsor) confirmed

Alex the Entreprenerd (judge) commented:

> `notSameBlock` is effectively being used as the `nonReentrant` modifier, without the same security guarantees, as such, in spite of not having a specific attack vector, because the warden showed how to side step this security feature of the protocol, am going to raise the severity to Medium

[M-06] AbstractRewardMine - Re-entrancy attack during withdrawal

Submitted by ScopeLift

The internal `_withdraw` method does not follow the checks-effects-interactions pattern. A malicious token, or one that implemented transfer hooks, could re-enter the public calling function (such as `withdraw()`) before proper internal accounting was completed. Because the `earned` function looks up the `_userWithdrawn` mapping, which is not yet updated when the transfer occurs, it would be possible for a malicious contract to re-enter `_withdraw` repeatedly and drain the pool.

Recommended Mitigation Steps

The internal accounting should be done before the transfer occurs:

```
    function _withdraw(address account, uint256 amountReward, addres
        _userWithdrawn[account] += amountReward;
        _globalWithdrawn += amountReward;f

       rewardToken.safeTransfer(to, amountReward);

        emit Withdraw(account, amountReward, to);
      }
```

[OxScotch (sponsor) confirmed]

[Alex the Entreprenerd (judge) commented]:

> The warden identified a re-entrancy vulnerability that, given the right token would allow to drain the entirety of the contract.

> Tokens with hooks (ERC777 and ERC677) would allow to exploit the contract and drain it in it's entirety.

> This is a very serious vulnerability. However it can happen exclusively on a malicious or a token with hooks, as such (while I recommend the sponsor to mitigate by following recommendation by the warden), the attack can be completely prevented by using a token without hooks.

> For that reason I'll rate the finding of medium severity (as it requires external conditions)

[M-07] `MovingAverage.setSampleMemory()` may broke MovingAverage, making the value of `exchangeRate` in `StabilizerNode.stabilize()` being extremely wrong

Submitted by WatchPug

https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/MovingAverage.sol#L424-L442

```
  function setSampleMemory(uint256 _sampleMemory)
    external
    onlyRole(ADMIN_ROLE, "Must have admin privs")
  {
    require(_sampleMemory > 0, "Cannot have sample memroy of 0");

    if (_sampleMemory > sampleMemory) {
      for (uint i = sampleMemory; i < _sampleMemory; i++) {
        samples.push();
      }
      counter = counter % _sampleMemory;
    } else {
      activeSamples = _sampleMemory;

      // TODO handle when list is smaller Tue 21 Sep 2021 22:29:41
    }

    sampleMemory = _sampleMemory;
  }
```

In the current implementation, when `sampleMemory` is updated, the samples index will be malposition, making `getValueWithLookback()` get the wrong samples, so that returns the wrong value.

Proof of Concept

- When initial sampleMemory is `10`
- After `movingAverage.update(1e18)` being called for 120 times
- The admin calls `movingAverage.setSampleMemory(118)` and set sampleMemory to `118`

The current `movingAverage.getValueWithLookback(sampleLength * 10)` returns `0.00000203312 e18`, while it’s expeceted to be `1e18`

After `setSampleMemory()`, `getValueWithLookback()` may also return `0` or revert FullMath: FULLDIV_OVERFLOW at L134.

Recommendation

Consider removing `setSampleMemory` function.

[0xScotch (sponsor) confirmed]

[Alex the Entreprenerd (judge) commented]:

> I agree that calling `setSampleMemory` will cause issues, and can cause opportunities to further extract value. However this can be triggered by an admin action. I'll think about the severity, but as of now, because it is contingent on admin privilege, will downgrade to Medium

[Alex the Entreprenerd (judge) commented]:

> I stand by my decision of Medium Severity. While the consequences can be troublesome, they are contingent on the admin breaking / griefing the system

[M-08] `_getFirstSample` returns wrong sample if count < sampleMemory

Submitted by cmichel

The `MovingAverage.sol` contract defines several variables that in the end make the `samples` array act as a ring buffer:

- `sampleMemory`: The total length (buffer size) of the `samples` array. `samples` is initialized with `sampleMemory` zero observations.
- `counter`: The pending sample index (modulo `sampleMemory`)

The `_getFirstSample` function computes the first sample as `(counter + 1) % sampleMemory` which returns the correct index only *if the ring buffer is full*, i.e., it wraps around. (in the `counter + 1 >= sampleMemory`).

If the `samples` array does not wrap around yet, the zero index should be returned instead.

Impact

Returning `counter + 1` if `counter + 1 < sampleMemory` returns a zero initialized `samples` observation index. This then leads to a wrong computation of the TWAP.

Recommended Mitigation Steps

Add an additional check for `if (counter + 1 < sampleMemory) return 0` in `_getFirstSample`.

OxScotch (sponsor) confirmed and disagreed with severity:

> Funds aren't directly at risk. I believe this is medium severity

Alex the Entreprenerd (judge) commented:

> Personally am not sure the first sample after wrapping would be `counter + 1` (why not `counter % sampleMemory`)

> That said, I do agree that before wrapping, the first item in the array is at the 0 index

> I agree that the protocol is not behaving properly so I can understand the high severity, that said I also agree with the Sponsor's statement, the worst case would be a leak of value, so Medium severity seems the most appropriate

[M-09] `UniswapHandler.maltMarketPrice` returns wrong decimals

Submitted by cmichel, also found by gzeon

The `UniswapHandler.maltMarketPrice` function returns a tuple of the `price` and the `decimals` of the price. However, the returned `decimals` do not match the computed `price` for the `else if (rewardDecimals < maltDecimals)` branch:

```
    else if (rewardDecimals < maltDecimals) {
      uint256 diff = maltDecimals - rewardDecimals;
      price = (rewardReserves.mul(10**diff)).mul(10**rewardDecimals)
      decimals = maltDecimals;
    }
```

Note that `rewardReserves` are in reward token decimals, `maltReserves` is a malt balance amount (18 decimals). Then, the returned amount is in `rewardDecimals + diffDecimals + rewardDecimals - maltDecimals = maltDecimals + rewardDecimals - maltDecimals = rewardDecimals`. However `decimals = maltDecimals` is wrongly returned.

Impact

Callers to this function will receive a price in unexpected decimals and might inflate or deflate the actual amount. Luckily, the `AuctionEscapeHatch` decides to completely ignore the returned `decimals` and as all prices are effectively in `rewardDecimals`, even if stated in `maltDecimals`, it currently does not seem to lead to an issue.

Recommendation

Fix the function by returning `rewardDecimals` instead of `maltDecimals` in the `rewardDecimals < maltDecimals` branch.

0xScotch (sponsor) confirmed

Alex the Entreprenerd (judge) commented:

> Finding is valid, because the returned value is unused, I agree with the medium severity

[M-10] AuctionParticipant.sol: `setReplenishingIndex` mistake could freeze unclaimed tokens

Submitted by harleythedog

In AuctionParticipant.sol, the function `setReplenishingIndex` is an admin function that allows manually setting `replenishingIndex`. As I have shown in my two previous findings, I believe that this function could be called frequently. In my opinion (and Murphy's law would agree), this implies that eventually an admin will accidentally set `replenishingIndex` incorrectly with this function.

Right now, `setReplenishingIndex` does not allow the admin to set `replenishingIndex` to a value smaller than it currently is. So, if an admin were to accidentally set this value too high, then it would be impossible to set it back to a lower value (the higher the value set, the worse this issue). All of the unclaimed tokens on auctions at smaller indices would be locked forever.

Proof of Concept

See code for `setReplenishingIndex` here: https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/AuctionParticipant.sol#L132

Recommended Mitigation Steps

Remove the require statement on line 136, so that an admin can set the index to a smaller value.

0xScotch (sponsor) confirmed

Alex the Entreprenerd (judge) commented:

> Agree with the finding in that if the `ADMIN` were to increase the `replenishingIndex` then the unclaim tokens at auctions below the index wouldn't be claimable anymore.

> I believe the warden properly highlighted what an hypothetical abuse of `admin privilege` would look like. As such I'll rate the finding with medium severity.

> I don't necessarily agree with the warden recommended mitigation, it may actually be best to simply delete the setter, or force it to go up by one index at a time after checking that all tokens are claimed

[M-11] No max for advanceIncentive

Submitted by gpersoon, also found by 0x1f8b

The function setAdvanceIncentive of DAO.sol doesn't check for a maximum value of incentive. If incentivewould be very large, then advanceIncentive would be very large and the function advance() would mint a large amount of malt.

The function setAdvanceIncentive() can only be called by an admin, but a mistake could be made. Also if an admin would want to do a rug pull, this would be an ideal place to do it.

Proof of Concept

https://github.com/code-423n4/2021-11-malt/blob/d3f6a57ba6694b47389b16d9d0a36a956c5e6a94/src/contracts/DAO.sol#L98-L104

```
    function setAdvanceIncentive(uint256 incentive)  externalonlyRol
      ...
      advanceIncentive = incentive;
```

https://github.com/code-423n4/2021-11-malt/blob/d3f6a57ba6694b47389b16d9d0a36a956c5e6a94/src/contracts/DAO.sol#L55-L63

```
    function advance() external {
    ...
      malt.mint(msg.sender, advanceIncentive * 1e18);
```

Recommended Mitigation Steps

Check for a reasonable maximum value in advance()

0xScotch (sponsor) confirmed and disagreed with severity:

> Definitely need to guard against arbitrarily large incentives. Disagree the risk is medium though.

Alex the Entreprenerd (judge) commented:

> Agree with the finding, this is an example of admin privilege, where the admin can set a variable which can be used to dilute the token and rug the protocol.

> Because this is contingent on the admin's action, I believe medium severity to be proper

Alex the Entreprenerd (judge) commented:

> The simple rationale on the medium severity is that the owner could set the incentive to an exorbitant amount with the goal of minting a lot of tokens for an exit scam

[M-12] Permissions - return values not checked when sending ETH

Submitted by ScopeLift, also found by nathaniel

On lines 85 and 101, ETH is transferred using a `.call` to an address provided as an input, but there is no verification that the call call succeeded. This can result in a call to `emergencyWithdrawGAS` or `partialWithdrawGAS` appearing successful but in reality it failed. This can happen when the provided `destination` address is a contract that cannot receive ETH, or if the `amount` provided is larger than the contract's balance

Proof of Concept

Enter the following in remix, deploy the `Receiver` contract, and send 1 ETH when deploying the `Permissions` contract. Call `emergencyWithdrawGAS` with the receiver address and you'll see it reverts. This would not be caught in the current code

```
    pragma solidity ^0.8.0;

    contract Receivier{}

    contract Permissions {
```

```
    constructor() payable {}

    function emergencyWithdrawGAS(address payable destination) ext
      (bool ok, ) = destination.call{value: address(this).balance}
      require(ok, "call failed");
    }
  }
```

Tools Used

Remix

Recommended Mitigation Steps

In `emergencyWithdrawGAS`:

```
- destination.call{value: address(this).balance}('');
+ (bool ok, ) = destination.call{value: address(this).balance}('
+ require(ok, "call failed");
```

And similar for `partialWithdrawGAS`

0xScotch (sponsor) confirmed

Alex the Entreprenerd (judge) commented:

> Agree with the finding, I believe if a developer were to not use safeTransfer we'd rate as medium, so while I believe the impact to be minimal (no composability), I'll keep the severity to medium

[M-13] Reducing the epoch length results in leaking value from advancement incentives

Submitted by TomFrench

Unintended advancement incentives being paid out to third party

Proof of Concept

`DAO.sol` incentives outside parties to advance the epoch by minting 100 MALT tokens for calling the `advance` function. This is limited by checking that the start timestamp of the next epoch has passed.

https://github.com/code-423n4/2021-11-malt/blob/d3f6a57ba6694b47389b16d9d0a36a956c5e6a94/src/contracts/DAO.sol#L55-L63

This start timestamp is calculated by multiplying the new epoch number by the length of an epoch and adding it to the genesis timestamp.

https://github.com/code-423n4/2021-11-malt/blob/d3f6a57ba6694b47389b16d9d0a36a956c5e6a94/src/contracts/DAO.sol#L65-L67

This method makes no accommodation for the fact that previous epochs may have been set to be a different length to what they are currently.

https://github.com/code-423n4/2021-11-malt/blob/d3f6a57ba6694b47389b16d9d0a36a956c5e6a94/src/contracts/DAO.sol#L111-L114

In the case where the epoch length is reduced, `DAO` will think that the epoch number can be incremented potentially many times. Provided the `advanceIncentive` is worth more than the gas necessary to advance the epoch will be rapidly advanced potentially many times paying out unnecessary incentives.

Recommended Mitigation Steps

Rather than calculating from the genesis timestamp, store the last time that the epoch length was modified and calculate from there.

[0xScotch (sponsor) confirmed]()

[Alex the Entreprenerd (judge) commented]():

> Given a specific epoch length, the system will be able to determine which incentives to pay out. Because the math for calculating the next epoch is based on

> the initial time, changing the `epochLength` can cause unintended consequences and allow for further calls to `advance` with the goal of receiving more caller incentives.

> The exploit can be triggered by admin privileges (changing epochLength), and because it's a leak of value, I agree with medium severity

[M-14] Wrong permissions on `reassignGlobalAdmin`

Submitted by cmichel

The `Permissions.reassignGlobalAdmin` function is supposed to only be run with the `TIMELOCK_ROLE` role, see `onlyRole(TIMELOCK_ROLE, "Only timelock can assign roles")`.

However, the `TIMELOCK_ROLE` is not the admin of all the reassigned roles and the `revokeRole(role, oldAccount)` calls will fail as it requires the `ADMIN_ROLE`.

Recommended Mitigation Steps

The idea might have been that only the `TIMELOCK` should be able to call this function, and usually it is also an admin, but the function strictly does not work if the caller *only* has the `TIMELOCK` roll and will revert in this case. Maybe governance decided to remove the admin role from the Timelock, which makes it impossible to call `reassignGlobalAdmin` anymore as both the timelock and admin are locked out.

[OxScotch (sponsor) confirmed]()

[Alex the Entreprenerd (judge) commented]():

> The warden has identified a flaw in the roles implementation, while the system seems to work when the timelock has multiple roles, the name of the roles implies a different functionality than what can actually be done. The sponsor confirms. While I believe the impact in the demo setup to be fairly minor, because the finding shows a flow in the role setup, and the sponsor confirmed, I agree with medium severity

[M-15] Bonding doesn't work with fee-on transfer tokens

Submitted by cmichel

Certain ERC20 tokens make modifications to their ERC20's `transfer` or `balanceOf` functions. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`.

Impact

The `Bonding._bond()` function will revert in the `_balanceCheck` when transferring a fee-on-transfer token as it assumes the entire `amount` was received.

Recommended Mitigation Steps

To support fee-on-transfer tokens, measure the asset change right before and after the asset-transferring calls and use the difference as the actual bonded amount.

0xScotch (sponsor) confirmed

Alex the Entreprenerd (judge) commented:

> Agree with the finding, the check will revert on a token that takes fees as the system assumes that `amount` is the amount that will be received

[M-16] theft of system profit

Submitted by danb, also found by leastwood and WatchPug

System profit comes from the stabilize function when the price of malt is above 1. Therefore it should not be allowed for anyone to take a part of the system profit when the price is above one. Right now, anyone can take a part of the investors' profits, even if they don't own any malt at all.

Proof of Concept

suppose that the price went up to 1.2 dai; the investors should get the reward for this rise. instead, anyone can take a part of the reward in the following steps: The price of malt is now 1.2 dai. Take a flash loan of a large amount of malt. Sell the malt for dai. Now the price went down to 1.1 dai because of the enormous swap. Call stabilize in order to lower the price to 1 dai. Buy malt to repay the flash loan at a lower cost with the bought dai. Repay the flash loan, and take the profit.

It is a sandwich attack because they sold malt for a high price, then they called stabilize to lower the value, then repurchase it for a low price.

The user made a profit at the expense of the investors.

If a user already has malt, it's even easier: just sell all malt at a high cost.

Recommended Mitigation Steps

in _beforeTokenTransfer, if the price is above 1$ and the receiver is the AMM pool, stabilize it.

Alex the Entreprenerd (judge) commented:

> @0xScotch The warden says that if the price is above the peg price, the system should stabilize it From reading the `PoolTransferVerification.verifyTransfer` , the system wants you to trade when Malt is above peg (minus tolerance)

> Would you say this finding is invalid?

0xScotch (sponsor) commented:

> @0xScotch The warden says that if the price is above the peg price, the system should stabilize it From reading the `PoolTransferVerification.verifyTransfer` , the system wants you to trade when Malt is above peg (minus tolerance)

> Would you say this finding is invalid?

> I think the issue being pointed out is that the stabilization above peg can be sandwiched and some profit that would go to LPs can be extracted. I think this is an issue but its not absolutely critical.

> We have discussed internally how to deal with this and the suggestion of calling stabilize would require some additional logic in the transfer verification as it would create a circular execution path.

1. User tries to sell at 1.2
2. Transfer verifier triggers stabilize
3. StabilizerNode tries to sell Malt ahead of initial user
4. Transfer verifier runs again and will call stabilize again unless we modify it to deal with this case.

> I don't think that is an issue but we will consider how to best move ahead with it.

Alex the Entreprenerd (judge) commented:

> Let's dissect the warden's finding:

```
The price of malt is now 1.2 dai.
Take a flash loan of a large amount of malt.
Sell the malt for dai.
Now the price went down to 1.1 dai because of the enormous swap.
Call stabilize in order to lower the price to 1 dai.
Buy malt to repay the flash loan at a lower cost with the bought
Repay the flash loan, and take the profit.
```

> -> Flashloan Malt (assumes liquidity to do so, but let's allow that) -> Sell malt for dai -> price goes lower -> Call stabilize -> system reduces price even further -> Buy back the malt -> repay the loan

> I feel like this is something that can't really be avoided as due to the permissionless nature of liquidity pools, limiting the ability to sell or buy to a path will be sidestepped by having other pools being deployed to avoid those checks.

> That said, the warden has identified a way to leak value from the price control system.

> To be precise the value is being extracted against traders, as to get Malt to 1.2 you'd need a trader to be so aggressive in their purchase to push the price that high. The warden showed how any arber / frontrunner can then "steal" the potential profits of the malt system users by frontrunning them.

> Due to the profit extraction nature of the finding, I believe Medium Severity to be correct

[M-17] Auction collateralToken won't work if token is fee-on-transfer token

Submitted by harleythedog

There are several ERC20 tokens that take a small fee on transfers/transferFroms (known as "fee-on-transfer" tokens). Most notably, USDT is an ERC20 token that has togglable transfer fees, but for now the fee is set to 0 (see the contract here: https://etherscan.io/address/0xdAC17F958D2ee523a2206206994597C13D831ec7#code). For these tokens, it should not be assumed that if you transfer `x` tokens to an address, that the address actually receives `x` tokens. In the current test environment, DAI is the only `collateralToken` available, so there are no issues. However, it has been noted that more pools will be added in the future, so special care will need to be taken if fee-on-transfer tokens (like USDT) are planned to be used as `collateralTokens`.

For example, consider the function `purchaseArbitrageTokens` in Auction.sol. This function transfers `realCommitment` amount of `collateralToken` to the liquidityExtension, and then calls `purchaseAndBurn(realCommitment)` on the liquidityExtension. The very first line of `purchaseAndBurn(amount)` is `require(collateralToken.balanceOf(address(this)) >= amount, "Insufficient balance");`. In the case of fee-on-transfer tokens, this line will revert due to the small fee taken. This means that all calls to `purchaseArbitrageTokens` will fail, which would be very bad when the price goes below peg, since no one would be able to participate in this auction.

Proof of Concept

See `purchaseArbitrageTokens` here: https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/Auction.sol#L177

See `purchaseAndBurn` here: https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/LiquidityExtension.sol#L117

Recommended Mitigation Steps

Add logic to transfers/transferFroms to calculate exactly how many tokens were actually sent to a specific address. In the example given with `purchaseArbitrageTokens`, instead of calling `purchaseAndBurn` with `realCommitment`, the contract should use the difference in the liquidityExtension balance after the transfer minus the liquidityExtension balance before the transfer.

[0xScotch (sponsor) confirmed]

[Alex the Entreprenerd (judge) commented]:

> Agree with the finding, the system as a whole seem to not deal with feeOnTransfer Token Mitigation can be as simple as never using them, or refactoring to check for actual amounts received

[M-18] AuctionParticipant.sol: `purchaseArbitrageTokens` should not push duplicate auctions

Submitted by harleythedog

In AuctionParticpant.sol, every time `purchaseArbitrageTokens` is called, the current auction is pushed to `auctionIds`. If this function were to be called on the same auction multiple times, then the same auction id would be pushed multiple times into this array, and the `claim` function would have issues with `replenishingIndex`.

Specifically, even if `replenishingIndex` was incremented once in `claim`, it is still possible that the auction at the next index will never reward any more tokens to the participant, so the contract would need manual intervention to set `replenishingIndex` (due to the if statement on lines 79-82 that does nothing if there is no claimable yield).

It is likely that `purchaseArbitrageTokens` would be called multiple times on the same auction. In fact, the commented out code for `handleDeficit` (in ImpliedCollateralService.sol) even suggests that the purchases might happen within the same transaction. So this issue will likely be an issue on most auctions and would require manual setting of `replenishingIndex`.

NOTE: This is a separate issue from the one I just submitted previously relating to `replenishingIndex`. The previous issue was related to an edge case where `replenishingIndex` might need to be incremented by one if there are never going to be more claims, while this issue is due to duplicate auction ids.

Proof of Concept

See code for `purchaseArbitrageTokens` here: https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/AuctionParticipant.sol#L40

Notice that `currentAuction` is always appended to `auctionIds`.

Recommended Mitigation Steps

Add a check to the function to `purchaseArbitrageTokens` to ensure that duplicate ids are not added. For example, this can be achieved by changing auctionIds to a mapping instead of an array.

[0xScotch (sponsor) confirmed]()

[Alex the Entreprenerd (judge) commented]():

> I agree with the finding, adding the same `auctionId` can lead to undefined behaviour. This can break claiming of incentives. Because of that, I think medium severity to be appropriate

[M-19] MiningService.setBonding should use BONDING role instead of REINVESTOR one

Submitted by hyh

BONDING_ROLE cannot be managed after it was initialized.

Proof of Concept

`setBonding` set the wrong role via _swapRole:

https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/MiningService.sol#L116

Recommended Mitigation Steps

Set `BONDING_ROLE` instead of `REINVESTOR_ROLE` in `setBonding` function:

Now:

```
    function setBonding(address _bonding)
      public
      onlyRole(ADMIN_ROLE, "Must have admin privs")
    {
      require(_bonding != address(0), "Cannot use address 0");
      _swapRole(_bonding, bonding, REINVESTOR_ROLE);
      bonding = _bonding;
    }
```

To be:

```
    function setBonding(address _bonding)
      public
      onlyRole(ADMIN_ROLE, "Must have admin privs")
    {
      require(_bonding != address(0), "Cannot use address 0");
      _swapRole(_bonding, bonding, BONDING_ROLE);
      bonding = _bonding;
    }
```

[0xScotch (sponsor) confirmed]()

[M-20] Users Can Contribute To An Auction Without Directly Committing Collateral Tokens

Submitted by leastwood

`purchaseArbitrageTokens` enables users to commit collateral tokens and in return receive arbitrage tokens which are redeemable in the future for Malt tokens. Each auction specifies a commitment cap which when reached, prevents users from participating in the auction. However, `realCommitment` can be ignored by directly sending the `LiquidityExtension` contract collateral tokens and subsequently calling `purchaseArbitrageTokens`.

Proof of Concept

Consider the following scenario:

- An auction is currently active.
- A user sends collateral tokens to the `LiquidityExtension` contract.
- The same user calls `purchaseArbitrageTokens` with amount `0`.
- The `purchaseAndBurn` call returns a positive `purchased` amount which is subsequently used in auction calculations.

As a result, a user could effectively influence the average malt price used throughout the `Auction` contract.

https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Auction.sol#L177-L214 https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/LiquidityExtension.sol#L117-L128

Recommended Mitigation Steps

Consider adding a check to ensure that `realCommitment != 0` in `purchaseArbitrageTokens`.

0xScotch (sponsor) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified a way to side-step the cap on commitments. Because the commitments are used for calculating limits, but `maltPurchased` is used to calculate rewards, an exploiter can effectively use an auction to purchase as many arbitrage tokens as they desire.

> Using any `amount` greater than zero will eventually allow to end the auction, however, by using 0 this process can be repeated continuously.

> Agree with the finding and severity

[M-21] `StabilizerNode` Will Mint An Incentive For Triggering An Auction Even If An Auction Exists Already

Submitted by leastwood

`_startAuction` utilises the `SwingTrader` contract to purchase Malt. If `SwingTrader` has insufficient capital to return the price of Malt back to its target price, an auction is triggered with the remaining amount. However, no auction is triggered if the current auction exists, but `msg.sender` is still rewarded for their call to `stabilize`.

Proof of Concept

`_shouldAdjustSupply` initially checks if the current auction is active, however, it does not check if the current auction exists. There is a key distinction between the `auctionActive` and `auctionExists` functions which are not used consistently. Hence, an auction which is inactive but exists would satisfy the edge case and result in `triggerAuction` simply returning.

- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Auction.sol#L382-L386
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Auction.sol#L268-L272
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/StabilizerNode.sol#L342-L344
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Auction.sol#L873-L888

Recommended Mitigation Steps

Consider using `auctionExists` and `auctionActive` consistently in `StabilizerNode` and `Auction` to ensure this edge case cannot be abused.

[0xScotch (sponsor) confirmed]

[Alex the Entreprenerd (judge) commented]:

> The warden found a way to effectively mint free malt by spamming the caller incentive by abusing a specific edge case.

[M-22] `_calculateMaltRequiredForExit` Uses Spot Price To Calculate Malt Quantity In `exitEarly`

Submitted by leastwood

`_calculateMaltRequiredForExit` in `AuctionEscapeHatch` currently uses Malt's spot price to calculate the quantity to return to the exiting user. This spot price simply tracks the Uniswap pool's reserves which can easily be manipulated via a flash loan attack to extract funds from the protocol.

Proof of Concept

- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/AuctionEscapeHatch.sol#L65-L92
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/AuctionEscapeHatch.sol#L193
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/DexHandlers/UniswapHandler.sol#L80-L109
- https://shouldiusespotpriceasmyoracle.com/

Recommended Mitigation Steps

Consider implementing/integrating a TWAP oracle to track the price of Malt.

[0xScotch (sponsor) confirmed]

[Alex the Entreprenerd (judge) commented]:

> I feel like this issue highlights the design challenge that the sponsor will have to solve, on one hand the protocol is meant to stabilize the price of malt in specific pools (impossible to block / control every pool due to permissionless nature). At

> the same time in order to determine which direction to move the price to, they need to refer to the pricing of the underlying pools (in this case a UniV2Pool, most likely from QuickSwap)

> Personally I understand the finding and think it's valid, however I don't believe there's easy answers as to how the sponsor should address this.

> Whenever there's excess value there will be entities trying to seize it and perhaps through such a harsh environment this protocol can truly find a way to be sustainable.

> That said, I'll mark the finding as valid, but believe this specific issue underlines the challenges that await the sponsor in making the protocol succesful

[M-23] `addLiquidity` Does Not Reset Approval If Not All Tokens Were Added To Liquidity Pool

Submitted by leastwood

`addLiquidity` is called when users reinvest their tokens through bonding events. The `RewardReinvestor` first transfers Malt and rewards tokens before adding liquidity to the token pool. `addLiquidity` provides protections against slippage by a margin of 5%, and any dust token amounts are transferred back to the caller. In this instance, the caller is the `RewardReinvestor` contract which further distributes the dust token amounts to the protocol's treasury. However, the token approval for this outcome is not handled properly. Dust approval amounts can accrue over time, leading to large Uniswap approval amounts by the `UniswapHandler` contract.

Proof of Concept

https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/DexHandlers/UniswapHandler.sol#L212-L214
https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/DexHandlers/UniswapHandler.sol#L216-L218

Recommended Mitigation Steps

Consider resetting the approval amount if either `maltUsed < maltBalance` or `rewardUsed < rewardBalance` in `addLiquidity`.

[0xScotch (sponsor) confirmed]

[Alex the Entreprenerd (judge) commented]:

> The UniV2Router will first [calculate the amounts] and then [pull them] from the msg.sender

> This means that approvals may not be fully utilized, leaving traces of approvals here and there. This can cause issues with certain tokens (USDT comes to mind), and will also not trigger gas refunds.

[M-24] `_distributeRewards` Does Not Reset Approval If Not All Tokens Were Allocated

Submitted by leastwood

`_distributeRewards` attempts to reward LP token holders when the price of Malt exceeds its price target. Malt Finance is able to being Malt back to its peg by selling Malt and distributing rewards tokens to LP token holders. An external call to `Auction` is made via the `allocateArbRewards` function. Prior to this call, the `StabilizerNode` approves the contract for a fixed amount of tokens, however, the `allocateArbRewards` function does not necessarily utilise this entire amount. Hence, dust token approval amounts may accrue from within the `StabilizerNode` contract.

Proof of Concept

https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/StabilizerNode.sol#L252-L253
https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Auction.sol#L809-L871

Recommended Mitigation Steps

Consider resetting the approval amount if the input `rewarded` amount to `allocateArbRewards` is less than the output amount.

[0xScotch (sponsor) confirmed]

[Alex the Entreprenerd (judge) commented]:

> Finding is valid, incorrect approvals can cause reverts (USDT being an example), and not fully resetting will also negate gas refunds

[M-25] AMM pool can be drained using a flashloan and calling `stabilize`

Submitted by stonesandtrees

All of the `rewardToken` in a given AMM pool can be removed from the AMM pool and distributed as LP rewards.

Proof of Concept

In the `stabilize` method in the `StabilizerNode` the initial check to see if the Malt price needs to be stabilized it uses a short period TWAP: https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/StabilizerNode.sol#L156

However, if the price is above the threshold for stabilization then the trade size required to stabilize looks at the AMM pool directly which is vulnerable to flashloan manipulation.

https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/DexHandlers/UniswapHandler.sol#L250-L275

Attack:

1. Wait for TWAP to rise above the stabilization threshold
2. Flashloan remove all but a tiny amount of Malt from the pool.

3. Call `stabilize`. This will pass the TWAP check and execute `_distributeSupply` which in turn ultimately calls `_calculateTradeSize` in the `UniswapHandler`. This calculation will determine that almost all of the `rewardToken` needs to be removed from the pool to return the price to peg.
4. Malt will mint enough Malt to remove a lot of the `rewardToken` from the pool.
5. The protocol will now distribute that received `rewardToken` as rewards. 0.3% of which goes directly to the attacker and the rest goes to LP rewards, swing trader and the treasury.

The amount of money that can be directly stolen by a malicious actor is small but it can cause a lot of pain for the protocol as the pool will be destroyed and confusion around rewards will be created.

Recommended Mitigation Steps

Use a short TWAP to calculate the trade size instead of reading directly from the pool.

0xScotch (sponsor) confirmed

Alex the Entreprenerd (judge) commented:

> I believe the warden has identified a valid grief and potential exploit

> I'm not convinced on the simplicity of: `2. Flashloan remove all but a tiny amount of Malt from the pool.`

> You'd have to buy that liquidity in order to be able to remove the malt, which effectively makes the operation not as straightforward (if not unprofitable for the attacker).

> I do believe the grief can be performed but in lack of a clear incentive for the attacker, am going to downgrade to Medium Severity. Can be done, but not clear on the incentives

[M-26] Dutch auction can be manipulated

Submitted by gzeon

When malt is under-peg and the swing trader module do not have enough capital to buy back to peg, a Dutch auction is triggered to sell arb token. The price of the Dutch auction decrease linearly toward endprice until _endAuction() is called. https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/Auction.sol#L589

_endAuction() is called in

1. When auction.commitments >= auction.maxCommitments

https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/Auction.sol#L212

2. On stabilize() -> checkAuctionFinalization() -> _checkAuctionFinalization()

https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/StabilizerNode.sol#L146 https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/Auction.sol#L754

3. On stabilize() ->_startAuction() -> triggerAuction() -> _checkAuctionFinalization()

https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/StabilizerNode.sol#L170 https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/Auction.sol#L754

It is possible manipulate the dutch auction by preventing _endAuction() being called.

Proof of Concept

Consider someone call purchaseArbitrageTokens with auction.maxCommitments minus 1 wei, `_endAuction` won't be called because auction.commitments < auction.maxCommitments. Further purchase would revert because `purchaseAndBurn` (https://github.com/code-423n4/2021-11-

malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/Auction.sol#L184) would likely revert since swapping 1 wei in most AMM will fail due to rounding error. Even if it does not revert, there is no incentive to waste gas to purchase 1 wei of token.

As such, the only way for the auction to finalize is to call stabilize(). However, this is not immediately possible because it require `block.timestamp >= stabilizeWindowEnd` where `stabilizeWindowEnd = block.timestamp + stabilizeBackoffPeriod` stabilizeBackoffPeriod is initially set to 5 minutes in the contract

After 5 minute, stabilize() can be called by anyone. By using this exploit, an attacker can guarantee he can purchase at (startingPrice+endingPrice)/2 or lower, given the default 10 minute auctionLength and 5 minute stabilizeBackoffPeriod. (unless a privileged user call stabilize() which override the stability window)

Also note that stabilize() might not be called since there is no incentive.

Recommended Mitigation Steps

1. Incentivize stabilize() or incentivize a permission-less call to _endAuction()
2. Lock-in auction price when user commit purchase

0xScotch (sponsor) labeled **sponsor confirmed**

Alex the Entreprenerd (judge) commented:

> By purchasing all but 1 wei of arbitrageTokens, any caller can guarantee that the auction will offer the steepest discount.

> This is caused by the fact that AMMs (esp UniV2) will revert with small numbers, as rounding will cause the amountOut to == 0 which will cause a INSUFFICIENT_AMOUNT revert.

> I agree with the pre-conditions and the possibility of this to happen. In a sense I believe this can become the meta strategy that every dutch auction participant will use (the fight between buyers will be done by paying gas to be the first few to buy arbitrageTokens).

> So the question is how to avoid it. I guess purchasing at a time should lock the price for that particular buyer at that particular time (adding a mapping should increase cost of 20k on write and a few thousand gas on read).

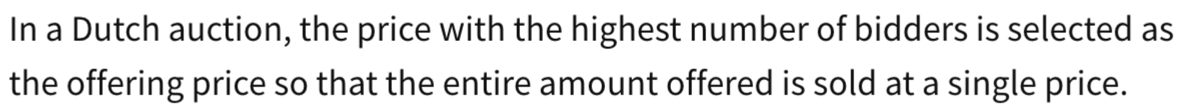
Alex the Entreprenerd (judge) commented:

> TODO: Decide on severity

> Medium for sure, extract value reliably. Is it high though? Arguably the dutch auction is not properly coded as it allows to get a further discount instead of locking in a price for the user at time of deposit

Alex the Entreprenerd (judge) commented:

> While an argument for the Dutch Auction being coded wrong (user locking in price) can be made, that's not the definition of a Dutch Auciton


In a Dutch auction, the price with the highest number of bidders is selected as the offering price so that the entire amount offered is sold at a single price.

https://www.investopedia.com/terms/d/dutchauction.asp

> So arguably the dutch auction is properly coded, it's that there's a specific way to sidestep it which is caused by:

- 1 wei tx reverting on swap
- sum of purchases not being enough

> For this conditions to happen the "exploiter" needs to be fast enough to place an order, in such a way that their order will leave 1 commitment left. I can imagine them setting up a contract that reverts if this condition isn't met and that checks for w/e amount is needed at that time.

> I would assume that having an admin privileged function to close the auction prematurely would solve this specific attack.

> I believe that the warden has identified a fairly reliable way to get a discount from the protocol because of the impact and some of the technicalities I believe Medium Severity to be more appropriate. This will be exploited, but in doing so will make the protocol successful (all auction full minus 1 wei), the premium / discount

> will be reliable predicted (50% between start and end) and as such I believe it will be priced in

[M-27] Slippage checks when adding liquidity are too strict

Submitted by cmichel

When adding liquidity through `UniswapHandler.addLiquidity`, the entire contract balances are used to add liquidity and the min amounts are set to 95% of these balances. If the balances in this contract are unbalanced (the ratio is not similar to the current Uniswap pool reserve ratios) then this function will revert and no liquidity is added.

See `UniswapHandler.buyMalt`:

```
(maltUsed, rewardUsed, liquidityCreated) = router.addLiquidity(
  address(malt),
  address(rewardToken),
  maltBalance, // @audit-info amountADesired
  rewardBalance,
  // @audit assumes that whatever is in this contract is already
  maltBalance.mul(95).div(100), // @audit-info amountAMin
  rewardBalance.mul(95).div(100),
  msg.sender, // transfer LP tokens to sender
  now
);
```

Impact

If the contract has unbalanced balances, then the `router.addLiquidity` call will revert. Note that an attacker could even send tokens to this contract to make them unbalanced and revert, resulting in a griefing attack.

Recommended Mitigation Steps

It needs to be ensured that the balances in the contract are always balanced and match the current reserve ratio. It might be better to avoid directly using the balances which can be manipulated by transferring tokens to the contract and accepting parameters instead of how many tokens to provide liquidity with from the caller side.

0xScotch (sponsor) confirmed

Alex the Entreprenerd (judge) commented:

> Interestingly the warden highlights the other side of the “missing slippage check” argument. Slippage checks in general need to be calculated offChain (as you will get frontrun in the mempool, so the slippage check is a risk minimization tool more than anything else)

> The warden also specified a griefing attack that can be used because of the hardcoded check. The sponsor confirms, I think medium severity is appropriate

[M-28] Bonding.sol _unbondAndBreak does not account for edge case where no tokens are returned

Submitted by harleythedog

In Bonding.sol, the internal function `_unbondAndBreak` transfers a user’s stake tokens to the dexHandler and then calls `removeLiquidity` on the dexHandler. Within the Uniswap handler (which is the only handler so far) `removeLiquidity` takes special care in the edge case where `router.removeLiquidity` returns zero tokens. Specifically, the Uniswap handler has this code:

```
if (amountMalt == 0 || amountReward == 0) {
  liquidityBalance = lpToken.balanceOf(address(this));
  lpToken.safeTransfer(msg.sender, liquidityBalance);
  return (amountMalt, amountReward);
}
```

If this edge case does indeed happen (i.e. if something is preventing the Uniswap router from removing liquidity at the moment), then the Uniswap handler will transfer the LP tokens back to Bonding.sol. However, Bonding.sol does not have any logic to recognize that this happened, so the LP tokens will become stuck in the contract and the user will never get any of their value back. This could be very bad if the user unbonds a lot of LP and they don’t get any of it back.

Proof of Concept

See `_unbondAndBreak` here: https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/Bonding.sol#L226

Notice how the edge case where `amountMalt == 0 || amountReward == 0` is not considered in this function, but it is considered in the Uniswap handler's `removeLiquidity` here: https://github.com/code-423n4/2021-11-malt/blob/c3a204a2c0f7c653c6c2dda9f4563fd1dc1cecf3/src/contracts/DexHandlers/UniswapHandler.sol#L240

Recommended Mitigation Steps

Add a similar edge case check to `_unbondAndBreak`. In the case where LP tokens are transferred back to Bonding.sol instead of malt/reward, these LP tokens should be forwarded back to the user since the value is rightfully theirs.

[0xScotch (sponsor) confirmed]()

[Alex the Entreprenerd (judge) commented]():

> In the `then` case the tokens will be sent back to the `Bonding.sol` contract, which has no way of rescuing or returning the tokens, probably reverting would be a better solution.

> Because the warden identified a way for tokens to get stucked, I think Medium Severity to be appropriate

[M-29] User can bypass Recovery Mode via UniswapHandler to buy Malt

Submitted by gzeon

One of the innovative feature of Malt is to block buying while under peg. The buy block can be bypassed by swapping to the whitelisted UniswapHandler, and then extract the token by abusing the add and remove liquidity function. This is considered a high severity issue because it undermine to protocol's ability to generate profit by the privileged role as designed and allow potential risk-free MEV.

Proof of Concept

1. User swap dai into malt and send malt directly to uniswapHandler, this is possible becuase uniswapHandler is whitelisted

`swapExactTokensForTokens(amountDai, 0, [dai.address, malt.address], uniswapHandler.address, new Date().getTime() + 10000);` 2) User send matching amount of dai to uniswapHandler 3) User call addLiquidity() and get back LP token 4) User call removeLiquidity() and get back both dai and malt

Recommended Mitigation Steps

According to documentation in https://github.com/code-423n4/2021-11-malt#high-level-overview-of-the-malt-protocol

> Users wanting to remove liquidity can still do so via the UniswapHandler contract that is whitelisted in recovery mode.

, this should be exploitable. Meanwhile the current implementation did not actually allow remove liquidity during recovery mode (refer to issue “Unable to remove liquidity in Recovery Mode”) This exploit can be mitigated by disabling addLiquidity() when the protocol is in recovery mode

0xScotch (sponsor) confirmed

Alex the Entreprenerd (judge) commented:

> The Malt token is programmed to explicitly prevent buying it when at discount. However, because `addLiquidity` and `removeLiquidity` are callable by anyone, and the `UniswapHandler` is whitelisted, through a calculated addition and removal of liquidity anyone can mimick buying Malt for a discount (by providing imbalanced underlying and redeeming them).

> I believe this sidesteps the majority of the functionality of the protocol, and while the attack is fairly simple, it denies the protocol functionality and as such agree with a High Severity

Alex the Entreprenerd (judge) commented:

> After re-review: Addressing each step:

1. The swap can be done as the whitelisting is bypassed (so finding is valid on that end)
2. User can send more liquidity
3. User can add liquidity at discount (as they bought at discount) (notice that this subjects them to potentially more IL so arguably the MEV extraction happened at step 1)
4. This cannot be done because of #323 which was found by the same warden. As per #323 you can't remove liquidity as the `transfer` will be from pool to user and as such will be reverted due to the system being in recovery mode.

> I believe that the warden identified a way to sidestep purchasing malt via purchase -> send to UniswapHandler -> add liquidity which should allow for some MEV extraction (with the user risking IL as they assume malt will go back to peg)

> Because of this there's still some validity to the finding, but it's not as dire as I originally believed.

> I'm going to downgrade the finding to Medium Severity as:

1. Value can be extracted
2. by bypassing the check for buying malt

> But this doesn't allow the selling of malt, so it's not a protocol breaking exploit but rather a way to gain MEV.

[M-30] Malt Protocol Uses Stale Results From `MaltDataLab` Which Can Be Abused By Users

Submitted by leastwood

`MaltDataLab` integrates several `MovingAverage` contracts to fetch sensitive data for the Malt protocol. Primary data used by the protocol consists of the real value for LP tokens, the average price for Malt and average reserve ratios. `trackMaltPrice`, `trackPoolReserves` and `trackPool` are called by a restricted role denoted as the `UPDATER_ROLE` and represented by an EOA account and not another contract. Hence, the EOA account must consistently update the aforementioned functions to ensure the most up-to-date values. However, miners can censor calls to

`MaltDataLab` and effectively extract value from other areas of the protocol which use stale values.

Proof of Concept

Consider the following attack vector:

- The price of Malt exceeds the lower bound threshold and hence `stabilize` can be called by any user.
- The `_stabilityWindowOverride` function is satisfied, hence the function will execute.
- The state variable, `exchangeRate`, queries `maltPriceAverage` which may use an outdated exchange rate.
- `_startAuction` is executed which rewards `msg.sender` with 100 Malt as an incentive for triggering an auction.
- As the price is not subsequently updated, a malicious attacker could collude with a miner to censor further pool updates and continue calling `stabilize` on every `fastAveragePeriod` interval to extract incentive payments.
- If the payments exceed what the `UPDATER_ROLE` is willing to pay to call `trackMaltPrice`, a user is able to sustain this attack.

This threatens the overall stability of the protocol and should be properly handled to prevent such attacks. However, the fact that `MaltDataLab` uses a series of spot price data points to calculate the `MovingAverage` also creates an area of concern as well-funded actors could still manipulate the `MovingAverage` contract by sandwiching calls to `trackMaltPrice`, `trackPool` and `trackPoolReserves`.

`trackMaltPrice`, `trackPool`, and `trackPoolReserves` should be added to the following areas of the code where applicable.

- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Bonding.sol#L159
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Bonding.sol#L173

- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Bonding.sol#L177
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Auction.sol#L881
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/Auction.sol#L710
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/StabilizerNode.sol#L156
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/StabilizerNode.sol#L190
- https://github.com/code-423n4/2021-11-malt/blob/main/src/contracts/ImpliedCollateralService.sol#L105

Recommended Mitigation Steps

Consider adding calls to `trackMaltPrice`, `trackPoolReserves` and `trackPool` wherever the values are impacted by the protocol. This should ensure the protocol is tracking the most up-to-date values. Assuming the cumulative values are used in the `MovingAverage` contracts, then sensitive calls utilising `MaltDataLab` should be protected from flashloan attacks. However, currently this is not the case, rather `MovingAverage` consists of a series of spot price data points which can be manipulated by well-funded actors or via a flashloan. Therefore, there needs to be necessary changes made to `MaltDataLab` to use cumulative price updates as its moving average instead of spot price.

0xScotch (sponsor) confirmed:

> Gas issues were the reason updates weren't inlined into paths that update critical values. However based on some thoughts from the team over the past few weeks and suggestions in other audit findings I think we can reduce gas enough to make it viable to inline it

Alex the Entreprenerd (judge) commented:

1. Miners can censor transactions, that is a fact
2. Relying on an external call can cause race conditions or situations where the call didn't happen (price didn't update in time)

3. Arguably a malicious actor could collude with miners with the goal of extracting further value

> However, the target chain for the deployment is Polygon, so unless they were to create custom validator code, and collude 1-1 with each validator (no flashbots on Polygon, yet) then the attack is increasingly more complex.

> The attack is reliant on the externalities of the validator + somebody incentivized enough to exploit this.

> I agree with he finding and believe after stabilize the system should auto-update the prices Because of the external requirements am downgrading to medium severity

Low Risk Findings (43)

- [L-01] sellMalt(), addLiquidity() and removeLiquidity() Allow Non Privileged Users Withdraw Fund *Submitted by Meta0xNull, also found by 0x1f8b*
- [L-02] Missing zero address check which will put forfeited rewards at risk(ForefeitHandler.sol) *Submitted by 0xwags*
- [L-03] Multiple Zero address transfer functions on contract Permissions *Submitted by BouSalman, also found by 0xwags and sabtikw*
- [L-04] DOS with unbounded loop *Submitted by Koustre*
- [L-05] setupParticipant() function does not check for zero address *Submitted by jayjonah8*
- [L-06] reassignGlobalAdmin() Lack of Zero Address Check *Submitted by Meta0xNull, also found by BouSalman*
- [L-07] Code does not match comments in "_finalizeAuction" (Auction.sol) *Submitted by ye0lde*
- [L-08] Revert transaction if it is unable to change data *Submitted by xYrYuYx*
- [L-09] Missing zero-address checks on contract initialization *Submitted by hyh, also found by 0x1f8b, cmichel, jayjonah8, tabish, and WatchPug*
- [L-10] The Contract Should safeApprove(0) first *Submitted by defsec, also found by hagrid and robee*
- [L-11] Lack of precision *Submitted by robee*

- [L-12] Incorrect error messages in `StabilizerNode.sol` *Submitted by pmerkleplant*
- [L-13] purchaseArbitrageTokens 0 amount *Submitted by pauliax, also found by jayjonah8*
- [L-14] `initialize` functions can be frontrun *Submitted by cmichel, also found by 0x1f8b, defsec, jayjonah8, leastwood, Meta0xNull, stonesandtrees, and WatchPug*
- [L-15] `StabilizerNode.sol` The current implementation is misconfiguration-prone for rewardToken with non-18 decimals *Submitted by WatchPug, also found by cmichel*
- [L-16] Auction.sol amendAccountParticipation has no zero division check *Submitted by harleythedog*
- [L-17] AuctionParticipant.sol: `replenishingIndex` incrementing should be improved *Submitted by harleythedog*
- [L-18] maxAmount and balance *Submitted by pauliax, also found by xYrYuYx*
- [L-19] Inclusive checks *Submitted by pauliax*
- [L-20] Inconsistencies when checking if the auction is active *Submitted by pauliax*
- [L-21] _setupRole() Deprecated and Not Using With Constructor Effectively Circumventing the Admin System *Submitted by Meta0xNull, also found by pauliax*
- [L-22] Validation of ‘to’ in transferAndCall and transferWithPermit *Submitted by pauliax*
- [L-23] DOMAIN_SEPARATOR can change *Submitted by pauliax, also found by leastwood*
- [L-24] Auction.claimArbitrage: if calculated claimable amount is too big, the remaining committed amount cannot be retrieved *Submitted by hyh*
- [L-25] Malt decimals inconsistency: StabilizerNode and DAO contracts use 18 as hard coded Malt decimals *Submitted by hyh*
- [L-26] setSampleMemory counter set to right value? *Submitted by gpersoon*
- [L-27] Lack Of Return Value Check On the Dex Handler Malt Price Calculation *Submitted by defsec*

- [L-28] SafeMath library is not always used in the contracts *Submitted by defsec*
- [L-29] `approve` return values not checked & unsafe *Submitted by cmichel, also found by defsec and WatchPug*
- [L-30] Missing Overflow Protection On the DeployedCapital *Submitted by defsec*
- [L-31] Deprecated Function Usage *Submitted by defsec*
- [L-32] Remove liquidity never ends up with left-over LP tokens *Submitted by cmichel*
- [L-33] `splitReinvest` does not provide liquidity at optimal ratio *Submitted by cmichel*
- [L-34] `totalDeclaredReward >= totalReleasedReward` not true in `AbstractRewardMine` *Submitted by cmichel*
- [L-35] Missing `maltDataLab.trackReserveRatio()` in some cases after `swingTrader.sellMalt()` *Submitted by WatchPug*
- [L-36] SafeMath Not Used Nearly At All In MovingAverage.sol *Submitted by jayjonah8*
- [L-37] Should include non-existing contract check *Submitted by jayjonah8*
- [L-38] reassignGlobalAdmin() Have No Transfer Ownership Pattern *Submitted by Meta0xNull*
- [L-39] The value of `reward` parameter of the `ProvideReinvest` event can be wrong *Submitted by WatchPug*
- [L-40] Max value of upperStabilityThreshold and lowerStabilityThreshold not checked *Submitted by gpersoon*
- [L-41] Adapt count in setAuctionAverageLookback? *Submitted by gpersoon, also found by WatchPug*
- [L-42] Unbounded loops *Submitted by pauliax*
- [L-43] Users may lose a small portion of promised returns due to precision loss *Submitted by WatchPug*

Non-Critical Findings (25)

- [N-01] Unused imports *Submitted by robee, also found by defsec, MetaOxNull, WatchPug, and xYrYuYx*
- [N-02] Implementations should inherit their interface *Submitted by WatchPug, also found by xYrYuYx*
- [N-03] deployedCapital variable is internal *Submitted by xYrYuYx*
- [N-04] Index address of events for better filtering *Submitted by tabish*
- [N-05] Use bps uniformly *Submitted by tabish*
- [N-06] Outdated Solidity Version Provides No Protections Against Arithmetic Underflows And Overflows *Submitted by leastwood, also found by 0x0x0x, BouSalman, hyh, sabtikw, and WatchPug*
- [N-07] Create2Deployer *Submitted by pauliax*
- [N-08] Inaccurate revert messages *Submitted by pauliax, also found by 0x0x0x, harleythedog, robee, and ScopeLift*
- [N-09] `setupParticipant` function should be internal *Submitted by nathaniel*
- [N-10] Outdated versions of OpenZeppelin library *Submitted by WatchPug*
- [N-11] Misleading variable names *Submitted by WatchPug*
- [N-12] Misleading error message *Submitted by WatchPug, also found by gpersoon and ScopeLift*
- [N-13] Race condition on ERC20 approval *Submitted by WatchPug, also found by robee*
- [N-14] Code Style: private/internal function names should be prefixed with `_` *Submitted by WatchPug*
- [N-15] No message in require statements. *Submitted by BouSalman*
- [N-16] functions visibility on contract AuctionEscapeHatch *Submitted by BouSalman*
- [N-17] functions visibility on contract Timelock.sol *Submitted by BouSalman*
- [N-18] Unused event or missed emit on SetAnnualYield() *Submitted by BouSalman*
- [N-19] Open TODOs *Submitted by yeOlde, also found by MetaOxNull, pauliax, and robee*
- [N-20] Missing events for admin only functions that change critical parameters *Submitted by defsec, also found by 0x0x0x, harleythedog, sabtikw,*

and WatchPug

- [N-21] governor or timelock *Submitted by gpersoon*
- [N-22] Wrong comment in `removeLiquidity` *Submitted by cmichel*
- [N-23] Initial `SetTransferService` event not emitted *Submitted by cmichel*
- [N-24] Permissions.sol#_swapRole is named wrongly *Submitted by 0x0x0x*
- [N-25] `permit` Double Emits An `Approval` Event *Submitted by leastwood*

Gas Optimizations (75)

- [G-01] Use short reason strings can save gas *Submitted by WatchPug, also found by GiveMeTestEther, Meta0xNull, robee, and yeOlde*
- [G-02] Cache array length in `for` loops *Submitted by pmerkleplant, also found by 0x0x0x, GiveMeTestEther, pauliax, robee, WatchPug, and yeOlde*
- [G-03] Cache Reference To State Variable "currentAuctionID" in _checkAuctionFinalization (Auction.sol) *Submitted by yeOlde, also found by harleythedog*
- [G-04] Assignment Of Variables To Default *Submitted by yeOlde, also found by 0x0x0x, GiveMeTestEther, and WatchPug*
- [G-05] Storage double reading. Could save SLOAD *Submitted by robee, also found by GiveMeTestEther, hyh, WatchPug, and yeOlde*
- [G-06] Underutilized Named Returns *Submitted by yeOlde*
- [G-07] Unused Named Returns *Submitted by yeOlde*
- [G-08] Unnecessary intermediate variables (MovingAverage.sol) *Submitted by yeOlde*
- [G-09] For uint `> 0` can be replaced with `!= 0` for gas optimisation *Submitted by 0x0x0x, also found by defsec, pmerkleplant, and yeOlde*
- [G-10] Unneeded variables (Auction.sol, StabilizerNode.sol) *Submitted by yeOlde*
- [G-11] Reduce external calls *Submitted by xYrYuYx*
- [G-12] decimals return of costBasis is not used. *Submitted by xYrYuYx*
- [G-13] Checking `uint256` variables `>= 0` is redundant *Submitted by WatchPug, also found by 0x0x0x, gzeon, pauliax, and xYrYuYx*

- [G-14] Cache decimals *Submitted by pauliax, also found by hyh and xYrYuYx*
- [G-15] In TransactionService, store index of source to avoid loop when removing verifier *Submitted by xYrYuYx*
- [G-16] Public functions to external *Submitted by robee, also found by 0x0x0x and xYrYuYx*
- [G-17] Storage Optimization *Submitted by 0x1f8b, also found by gzeon, robee, and tabish*
- [G-18] initialized storage variables are set again in the initializer function *Submitted by sabtikw*
- [G-19] Unused declared local variables *Submitted by robee, also found by Koustre*
- [G-20] Can remove treasuryRewardCut from ForfeitHandler.sol *Submitted by harleythedog, also found by pmerkleplant*
- [G-21] Don't try bonding zero liquidity in `RewardReinvestor` *Submitted by pmerkleplant*
- [G-22] (10000 - thresholdBps) can be pre-calculated *Submitted by pauliax*
- [G-23] Only use `SafeMath` when necessary can save gas *Submitted by WatchPug, also found by pauliax and ScopeLift*
- [G-24] Redundant require statements in `Auction:purchaseArbitrageTokens` *Submitted by loop, also found by pauliax*
- [G-25] ERC20 import *Submitted by pauliax*
- [G-26] Timelock reuse function argument as argument for the event emit *Submitted by GiveMeTestEther, also found by pauliax*
- [G-27] `++i` is more gas efficient than `i++` *Submitted by WatchPug, also found by defsec, jayjonah8, and pauliax*
- [G-28] Similar code in `getCollateralValueInMalt` and `totalUsefulCollateral` functions *Submitted by nathaniel*
- [G-29] Duplicated code in `unbond` and `unbondAndBreak` functions *Submitted by nathaniel*
- [G-30] Auction.userClaimableArbTokens nonzero auction.finalPrice check is redundant *Submitted by hyh, also found by nathaniel and shenwilly*
- [G-31] Redundant checks *Submitted by WatchPug, also found by loop*

- [G-32] SwingTrader.costBasis function should have internal version that uses Malt balance from sellMalt *Submitted by hyh*
- [G-33] SwingTrader: sellMalt and costBasis functions can be simplified *Submitted by hyh*
- [G-34] Auction.userClaimableArbTokens amountOut calculations can be simplified *Submitted by hyh*
- [G-35] Auction.userClaimableArbTokens claimablePerc calculations can be simplified *Submitted by hyh*
- [G-36] Unncessary statement in UniswapHandler.sol removeBuyer *Submitted by harleythedog*
- [G-37] Invalid equation check on `require` *Submitted by hagrid*
- [G-38] Gas optimization: Unnecessary return string *Submitted by gzeon*
- [G-39] Custom size uint is not more efficient than uint256 *Submitted by 0x0x0x, also found by cmichel and defsec*
- [G-40] Checking if `lpProfitCut > 0` can save gas *Submitted by WatchPug*
- [G-41] `MovingAverage.sol#_getFirstSample()` Implementation can be simpler and save some gas *Submitted by WatchPug*
- [G-42] AbstractRewardMine._handleStakePadding logic cases can be separated and function simplified *Submitted by hyh*
- [G-43] AbstractRewardMine._handleStakePadding calls totalDeclaredReward and this way balanceOf function twice *Submitted by hyh*
- [G-44] AbstractRewardMine.getRewardOwnershipFraction shouldn't be used internally *Submitted by hyh*
- [G-45] `AuctionBurnReserveSkew.sol#getRealBurnBudget()` Implementation can be simpler and save some gas *Submitted by WatchPug*
- [G-46] Returning the named returns is redundant *Submitted by WatchPug*
- [G-47] `AuctionBurnReserveSkew.sol#getPegDeltaFrequency()` Implementation can be simpler and save some gas *Submitted by WatchPug*
- [G-48] Cache external call results can save gas *Submitted by WatchPug*
- [G-49] Unnecessary internal function calls *Submitted by WatchPug*
- [G-50] Unused storage variables *Submitted by WatchPug*
- [G-51] Use immutable variable can save gas *Submitted by WatchPug*

- [G-52] `uint64(block.timestamp % 2**64)` can be simpler and save some gas *Submitted by WatchPug*
- [G-53] `MovingAverage.sol` Use inline expression can save gas *Submitted by WatchPug*
- [G-54] Unnecessary event fields *Submitted by TomFrench*
- [G-55] RewardReinvestor - safeTransfer used unnecessarily on Malt token *Submitted by ScopeLift*
- [G-56] Various contracts - remove unused function parameters to save gas *Submitted by ScopeLift*
- [G-57] Various contracts - stricter function mutability for gas savings *Submitted by ScopeLift*
- [G-58] AuctionBurnReserveSkew - remove `for` loop from initializer *Submitted by ScopeLift*
- [G-59] MovingAverage:getValueWithLookback move sampleDiff to save gas *Submitted by GiveMeTestEther*
- [G-75] removeVerifier() Repeat SLOAD During Loop Is Waste of Gas *Submitted by MetaOxNull*
- [G-60] MovingAverage:getValue move the declaration/initialization of sampleDiff to save gas in the case of an early return *Submitted by GiveMeTestEther*
- [G-74] Avoiding Initialization of Loop Index If It Is 0 *Submitted by MetaOxNull*
- [G-61] MovingAverage:initialize reuse argument variable instead storage variable in the loop condition *Submitted by GiveMeTestEther*
- [G-62] AuctionBurnReserveSkew:addAbovePegObservation gas optimization *Submitted by GiveMeTestEther*
- [G-63] AuctionBurnReserveSkew:getRealBurnBudget no underflow check needed *Submitted by GiveMeTestEther*
- [G-64] Auction:amendAccountParticipation no underflow check needed *Submitted by GiveMeTestEther*
- [G-65] Auction:claimArbitrage no underflow checks needed *Submitted by GiveMeTestEther*
- [G-66] Auction:purchaseArbitrageTokens gas optimization *Submitted by GiveMeTestEther*

- [G-67] RewardThrottle:handleReward gas optimizations *Submitted by GiveMeTestEther*
- [G-68] RewardDistributor:decrementRewards no underflow check needed *Submitted by GiveMeTestEther*
- [G-69] RewardDistributor:_incrementFocalPoint() save storage read *Submitted by GiveMeTestEther*
- [G-70] RewardDistributor:_forfeit no underflow check needed *Submitted by GiveMeTestEther*
- [G-71] Dont calculate progressionBps, when not needed *Submitted by 0x0x0x*
- [G-72] AbstractRewardMine.sol#_removeFromStakePadding can be implemented more efficiently *Submitted by 0x0x0x*
- [G-73] State variables that could be set immutable *Submitted by robee, also found by hyh*

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.