Mute Switch - Versus contest Findings & Analysis Report

2023-04-27

Table of contents

Overview

About C4

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

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

During the audit outlined in this document, C4 conducted an analysis of the Mute Switch smart contract system written in Solidity. The audit took place between March 28—April 3 2023.

Wardens

In Code4rena's Invitational audits, the competition is limited to a small group of wardens; for this audit, 5 wardens contributed reports:

1. 0xA5DF
2. HollaDieWaldfee
3. chaduke
4. evan
5. hansfriese

This audit was judged by Picodes.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 12 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 9 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 5 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 3 reports recommending gas optimizations.

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

Scope

The code under review can be found within the C4 Mute Switch audit repository, and is composed of 3 smart contracts written in the Solidity programming language and includes 535 lines of Solidity code.

Severity Criteria

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

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

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

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

High Risk Findings (3)

[H-01] Bond max-buyer might end up buying the max buy of the next epoch

Submitted by 0xA5DF, also found by hansfriese and HollaDieWaldfee

The `MuteBond.deposit()` function allows users to specify the amount of value they want to purchase bonds for or to set `max_buy` to true. If `max_buy` is set to true the amount specified in the `value` parameter is ignored and instead the maximum amount available for purchase in the current epoch is used. This can lead to a scenario where a user intends to purchase the remaining amount of current epoch, but till the tx is included in the blockchain a new epoch starts (either by an innocent user or by an attacker) and the user ends up buying the entire amount of the next epoch.

Impact

A. The user ends up buying a much higher amount than intended
B. The user ends up buying it for a lower price than intended (i.e. less payout for the

buyer)

Proof of Concept

The PoC below shows how `maxPurchaseAmount()` increases when a new era starts.

File: `test/bonds.ts`

```
    it('Max buy PoC', async function () {

      // buy 99% of amount available for purchase in current epocl
      let maxValue = await bondContract.maxPurchaseAmount();
      let depositValue = maxValue.mul(99).div(100);
      await bondContract.connect(buyer1).deposit(depositValue, buy

      // The amount available when the victim sends out the tx
      var expectedDeposit = await bondContract.maxPurchaseAmount()

      await bondContract.connect(buyer1).deposit('0', buyer1.addre

      // The amount available when the victims's tx is included ir
      var actualDeposit = await bondContract.maxPurchaseAmount();

      // expected deposit = 1 wad
      // actual deposit = 100 wad
      console.log({expectedDeposit, actualDeposit});
    })
```

The following snippet shows that when a user sets `max_buy` to true the value used is the `maxPurchaseAmount()`

```
        if(max_buy == true){
          value = maxPurchaseAmount();
          payout = maxDeposit();
        } else {
```

Recommended Mitigation Steps

Require the user to specify the epoch number when doing a ‘max buy’, and revert if it doesn’t match the current epoch (it might be a good idea to refactor the code to 2 external functions for normal buy and max buy, where they both share an internal function to make the actual deposit).

Side note: this is similar to another bug I've reported regarding getting a lower price than expected, however the root cause, impact, and mitigation are different and therefore I've reported this separately.

[mattt21 (Mute Switch) confirmed]()

[H-02] Attacker can front-run Bond buyer and make them buy it for a lower payout than expected

Submitted by [0xA5DF](), also found by HollaDieWaldfee ([1](), [2]()) and chaduke ([1](), [2](), [3]())

The `MuteBond` contract contains a feature in which after each purchase the `epochStart` increases by 5% of the time passed since `epochStart`, this (in most cases) lowers the bond's price (i.e. buyer gets less payout) for future purchases.

An attacker can exploit this feature to front-run a deposit/purchase tx and lower the victim's payout.

This can also happen by innocent users purchasing before the victim's tx is included in the blockchain.

Another (less likely) scenario in which this can happen is when the owner changes the config in a way that lowers the price (e.g. lowering max price, extending epoch duration), if the owner tx executes while a user's `deposit()` tx is in the mempool the user would end up with less payout than intended.

Side note: the term 'bond price' might be confusing since it refers to the payout the buyer gets divided by the value the buyer pays, so a higher price is actually in favor of the buyer.

Impact

User ends up buying bond for a lower payout than intended.

Proof of Concept

In the PoC below, an attacker manages to make the buyer purchase a bond at a price lower by 32% than intended.

File: `test/bonds.ts`

```
    it('Front run PoC', async function () {
      // let price reach the max price
      await time.increase(60 * 60 * 24 * 7)

      // price when victim sends out the tx to the mempool
      var expectedPrice = await bondContract.bondPrice()

      const startPrice = new BigNumber(100).times(Math.pow(10,18))
      let minPurchasePayout = new BigNumber(Math.pow(10,16));
      // using dynamic price didn't work out so I'm using the lowe
      var minPurchaseValue = minPurchasePayout.times(1e18).div(sta

      // attacker buys the lowest amount 20 times
      for(let i = 0; i< 20; i++){
        await bondContract.connect(buyer1).deposit(minPurchaseVal
      }

      var init_dmute = await dMuteToken.GetUnderlyingTokens(buyer1
      let depositValue = new BigNumber(10).times(Math.pow(10,18)).
      var price = await bondContract.connect(buyer1).deposit(depos
      var post_dmute = await dMuteToken.GetUnderlyingTokens(buyer1

      var dmute_diff = new BigNumber(post_dmute.toString()).minus(
      var actualPrice = dmute_diff.times(1e18).div(depositValue);

      var receipt = (await price.wait())
      // compare the expected price with the actual price
      // expected price = 200; actual price = 135.8; meaning actua
      console.log({expectedPrice, actualPrice:actualPrice.toString
    })
```

Recommended Mitigation Steps

Add a min payout parameter so that users can specify the expected payout. The tx should revert if the actual payout is lower than expected.

matttt21 (Mute Switch) confirmed

[H-03] `dMute.sol` : Attacker can push lock items to victim's array such that redemptions are forever blocked

Submitted by HollaDieWaldfee, *also found by* evan *and* chaduke

https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/dao/dMute.sol#L90-L129
https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/dao/dMute.sol#L135-L139

This report deals with how an attacker can abuse the fact that he can lock `MUTE` tokens for any other user and thereby push items to the array of `UserLockInfo` structs of the user.

There are two functions in the `dMute` contract that iterate over all items in this array (`RedeemTo` and `GetUnderlyingTokens`).

Thereby if the attacker pushes sufficient items to the array of a user, he can make the above two functions revert since they require more Gas than the Block Gas Limit.

According to the `zkSync` documentation the block gas limit is currently 12.5 million (Link).

The attack is of "High" impact for the `RedeemTo` function since this function needs to succeed in order for the user to redeem his `MUTE` tokens.

The user might have a lot of `MUTE` tokens locked and the attacker can make it such that they can never be redeemed. The attacker cannot gain a profit from this attack, i.e. he cannot steal anything, but due to the possibility of this attack users will not lock their tokens, especially not a lot of them.

This is all the more severe because the `MuteBond` and `MuteAmplifier` contracts also rely on the locking functionality so those upstream features can also not be used securely.

In the Mitigation section I will show how the `GetUnderlyingTokens` function can be made to run in $O(1)$ time instead of $O(lock:array:length)$.

The `RedeemTo` function can be made to run in $O(indexes:array:length)$ instead of $O(lock:array:length)$. The length of the indexes array is determined by the user and simply tells how many locked items to redeem. So there is no possibility of DOS.

Proof of Concept

Note: a redemption costs `~7 million Gas` when 1000 items are locked. So when running on the `zkSync` network even 2000 items should be enough. The hardhat tests use a local Ethereum network instead of a fork of `zkSync` so in order to hit `30 million Gas` (which is the Ethereum block gas limit) we need to add more items to the queue.

You can add the following test to the `dao.ts` test file:

```
it('Lock DOS', async function () {
    var tx = await muteToken.approve(dMuteToken.address, MaxUint


    let lock_time_week = new BigNumber(60 * 60 * 24 * 7);
    let max_lock = lock_time_week.times(52);

    let lock_amount = new BigNumber(1).times(Math.pow(10,2))

    // @audit fill up array
    for(let i=0;i<5000;i++) {
        tx = await dMuteToken.LockTo(lock_amount.toFixed(0), loc
    }

    await time.increase(60 * 60 * 24 * 7)

    tx = await dMuteToken.Redeem([0])
})
```

It adds `5000` lock items to the array of the `owner` address. When the `owner` then tries to redeem even a single lock the transaction fails due to an out of gas error.

(Sometimes it reverts with `TransactionExecutionError: Transaction ran out of gas` error sometimes it reverts due to timeout. If you try a few times it should revert with the out of gas error.)

The amount of `MUTE` tokens that the attacker loses to execute this attack is negligible. As you can see in the test `100 Wei * 5000 = 500,000 Wei` is sufficient (There needs to be some amount of `MUTE` such that the `LockTo` function does not revert). The only real cost comes down to Gas costs which are cheap on `zkSync`.

Tools Used

VSCode

Recommended Mitigation Steps

First for the `GetUnderlyingTokens` function: The contract should keep track of underlying token amounts for each user in a mapping that is updated with every lock / redeem call. The `GetUnderlyingTokens` function then simply needs to return the value from this mapping.

Secondly, fixing the issue with the `RedeemTo` function is a bit harder. I discussed this with the sponsor and I have been told they don't want this function to require an already sorted `lock_index` array as parameter. So the `lock_index` array can contain indexes in random order.

This means it must be sorted internally. Depending on the expected length of the `lock_index` array different sorting algorithms may be used. I recommend to use an algorithm like [quick sort](#) to allow for many indexes to be specified at once.

I will use a placeholder for the sorting algorithm for now so the sponsor may decide which one to use.

The proposed fixes for both functions are then like this:

```
diff --git a/contracts/dao/dMute.sol b/contracts/dao/dMute.sol
index 59f95b7..11d21fb 100644
--- a/contracts/dao/dMute.sol
+++ b/contracts/dao/dMute.sol
@@ -18,6 +18,7 @@ contract dMute is dSoulBound {
     }

     mapping(address => UserLockInfo[]) public _userLocks;
+    mapping(address => uint256) public _amounts;

     uint private unlocked = 1;

@@ -79,6 +80,7 @@ contract dMute is dSoulBound {
         _mint(to, tokens_to_mint);

         _userLocks[to].push(UserLockInfo(_amount, block.timesta
+        _amounts[to] = _amounts[to] + _amount;

         emit LockEvent(to, _amount, tokens_to_mint, _lock_time)
     }
@@ -91,8 +93,14 @@ contract dMute is dSoulBound {
         uint256 total_to_redeem = 0;
         uint256 total_to_burn = 0;

-        for(uint256 i; i < lock_index.length; i++){
-          uint256 index = lock_index[i];
+        ///////////////////////////////////////////////
+        //                                           //
+        // sort lock_index array in ascending order //
+        //                                          //
+        //////////////////////////////////////////////
+
+        for(uint256 i = lock_index.length; i > 0; i--){
+          uint256 index = lock_index[i - 1];
           UserLockInfo memory lock_info = _userLocks[msg.sender

           require(block.timestamp >= lock_info.time, "dMute::Re
@@ -102,23 +110,14 @@ contract dMute is dSoulBound {
           total_to_redeem = total_to_redeem.add(lock_info.amour
           total_to_burn = total_to_burn.add(lock_info.tokens_mi

-          _userLocks[msg.sender][index] = UserLockInfo(0,0,0);
+          _userLocks[msg.sender][index] = _userLocks[msg.sender
+          _userLocks[msg.sender].pop();
         }
```

```
        require(total_to_redeem > 0, "dMute::Lock: INSUFFICIENT
        require(total_to_burn > 0, "dMute::Lock: INSUFFICIENT_E

-
-       for(uint256 i = _userLocks[msg.sender].length; i > 0; i
-         UserLockInfo memory lock_info = _userLocks[msg.sender
-
-         // recently redeemed lock, destroy it
-         if(lock_info.time == 0){
-           _userLocks[msg.sender][i - 1] = _userLocks[msg.senc
-           _userLocks[msg.sender].pop();
-         }
-       }
-
+       _amounts[msg.sender] = _amounts[msg.sender] + total_to_
        //redeem tokens to user
        IERC20(MuteToken).transfer(to, total_to_redeem);
        //burn dMute
@@ -133,8 +132,6 @@ contract dMute is dSoulBound {
    }

    function GetUnderlyingTokens(address account) public view r
-       for(uint256 i; i < _userLocks[account].length; i++){
-         amount = amount.add(_userLocks[account][i].amount);
-       }
+       return _amounts[account];
    }
 }
```

Picodes (judge) commented:

> At first read, it looks like a great finding considering the project will be on `zkSync`.

mattt21 (Mute Switch) confirmed

Medium Risk Findings (9)

[M-01] Award is still distributed when there aren't any stakers, allowing users to get reward without staking

Submitted by evan, also found by HollaDieWaldfee

Consider the update modifier for the amplifier.

```
    modifier update() {
            if (_mostRecentValueCalcTime == 0) {
                _mostRecentValueCalcTime = firstStakeTime;
            }

            uint256 totalCurrentStake = totalStake();

            if (totalCurrentStake > 0 && _mostRecentValueCalcTime <
                uint256 value = 0;
                uint256 sinceLastCalc = block.timestamp.sub(_mostRec
                uint256 perSecondReward = totalRewards.div(endTime.s

                if (block.timestamp < endTime) {
                    value = sinceLastCalc.mul(perSecondReward);
                } else {
                    uint256 sinceEndTime = block.timestamp.sub(endTi
                    value = (sinceLastCalc.sub(sinceEndTime)).mul(pe
                }

                _totalWeight = _totalWeight.add(value.mul(10**18).di

                _mostRecentValueCalcTime = block.timestamp;

                (uint fee0, uint fee1) = IMuteSwitchPairDynamic(lpTo

                _totalWeightFee0 = _totalWeightFee0.add(fee0.mul(10*
                _totalWeightFee1 = _totalWeightFee1.add(fee1.mul(10*

                totalFees0 = totalFees0.add(fee0);
                totalFees1 = totalFees1.add(fee1);
            }

            _;
        }
```

Suppose there's been a period with totalCurrentStake = 0, and a user stakes and immediately withdraws in the same transaction.

When the user stakes, update doesn't do anything (including updating _mostRecentValueCalcTime) since totalCurrentStake = 0, and _userWeighted[account] gets set to _totalWeight (which hasn't been updated) in the stake function.

https://github.com/code-423n4/2023-03-mute/blob/main/contracts/amplifier/MuteAmplifier.sol#L349

```
    function _stake(uint256 lpTokenIn, address account) private {
            ...
            _userWeighted[account] = _totalWeight;
            ...
        }
```

When the user withdraws, totalCurrentStake is no longer zero. Since _mostRecentValueCalcTime wasn't updated when the user staked (since totalCurrentStake was 0), the reward from the period with no stakers gets added to _totalWeight.

```
 uint256 sinceLastCalc =
block.timestamp.sub(_mostRecentValueCalcTime);
 value = sinceLastCalc.mul(perSecondReward);
 _totalWeight =
_totalWeight.add(value.mul(10**18).div(totalCurrentStake));
```

(These are lines in the update modifier)

As a result, this user who staked and immediately withdrew, got all the reward from the period with no stakers. See the reward calculation: https://github.com/code-423n4/2023-03-mute/blob/main/contracts/amplifier/MuteAmplifier.sol#L371

```
    function _applyReward(address account) private returns (uint256
            ...
            reward = lpTokenOut.mul(_totalWeight.sub(_userWeighted[a
            ...
```

```
        }
```

Observe that the exploit condition is met as soon as the staking period starts (as long as nobody stakes immediately). The code attempts to prevent this situation setting _mostRecentValueCalcTime to firstStakeTime the first time that update is invoked (which must be a stake call). https://github.com/code-423n4/2023-03-mute/blob/main/contracts/amplifier/MuteAmplifier.sol#L89-L91

```
    if (_mostRecentValueCalcTime == 0) {
                _mostRecentValueCalcTime = firstStakeTime;
            }
```

However, this doesn't do anything since the user can first set the firstStakeTime by staking as soon as the staking period starts, and then make totalCurrentStake 0 by immediately withdrawing. In fact, observe that this takes away from other staker's rewards since perSecondReward is now lower. https://github.com/code-423n4/2023-03-mute/blob/main/contracts/amplifier/MuteAmplifier.sol#L98

```
 uint256 perSecondReward =
totalRewards.div(endTime.sub(firstStakeTime));
```

Please add the following test to the "advance to start time" context in amplifier.ts (add it here), and run with `npm run test-amplifier`.

```
    it("get reward without staking", async function () {
            await this.lpToken.transfer(this.staker1.address, staker
            await this.lpToken.connect(this.staker1).approve(
              this.amplifier.address,
              staker1Initial.toFixed()
            );

            console.log("dmute balance before: " + await this.dMuteT
            await this.amplifier.connect(this.staker1).stake(1);
            await this.amplifier.connect(this.staker1).withdraw();
            await time.increaseTo(staking_end-10);
            await this.amplifier.connect(this.staker1).stake(1);
            await this.amplifier.connect(this.staker1).withdraw();
            console.log("dmute balance after: " + await this.dMuteTo
          });
          it("get reward with staking", async function () {
            await this.lpToken.transfer(this.staker1.address, staker
            await this.lpToken.connect(this.staker1).approve(
              this.amplifier.address,
              staker1Initial.toFixed()
            );
            console.log("dmute balance before: " + await this.dMuteT
            await this.amplifier.connect(this.staker1).stake(1);
            //await this.amplifier.connect(this.staker1).withdraw();
            await time.increaseTo(staking_end-10);
            //await this.amplifier.connect(this.staker1).stake(1);
            await this.amplifier.connect(this.staker1).withdraw();
            console.log("dmute balance after: " + await this.dMuteTo
          });
```

Result on my side:

```
advance to start time
        ✓ reverts without tokens approved for staking
dmute balance before: 0
dmute balance after: 576922930658129099273
        ✓ get reward without staking
dmute balance before: 0
dmute balance after: 576922912276064160247
        ✓ get reward with staking
```

This POC is a bit on the extreme side to get the point across. In the first test, the user stakes and then immediately unstakes, while in the second test, the user stakes for the entire period. In the end, the user gets roughly the same amount of reward.

Impact

After periods with no stakers, users can get reward without staking. This is also possible at the beginning of the staking period, and doing so then will reduce the reward for other users in the process.

Tools Used

Manual review, Hardhat

Recommended Mitigation Steps

Possible solution 1: set a minimum duration that a user must stake for (prevent them from staking and immediately withdrawing)

Possible solution 2: always update _mostRecentValueCalcTime (regardless totalCurrentStake). i.e. move the following line out of the if statement.
https://github.com/code-423n4/2023-03-mute/blob/main/contracts/amplifier/MuteAmplifier.sol#L109

Keep in mind that with solution 2, no one gets the rewards in periods with no stakers - this means that the rescueTokens function needs to be updated to get retrieve these rewards.

mattt21 (Mute Switch) confirmed, but disagreed with severity

evan (warden) commented:

> Can the sponsor explain why they disagree with the severity?

Picodes (judge) commented:

> The issue is valid in my opinion. The desired behavior from this line seems to be that rewards not distributed because there is no stakers are added to the rewards per seconds distributed when there are some. The fact that the first next staker could take all pending rewards is a bug.

> Another mitigation could be to reset `firstStakeTime` and replicate the initial behavior if at some point `totalStake()` goes back to 0.

Picodes (judge) commented:

> In the absence of comment from the sponsor, I'll keep Medium severity, considering it isn't the desired behavior.

[M-02] A user can 'borrow' dMute balance for a single block to increase their amplifier APY

Submitted by 0xA5DF

The amplifier's APY is calculated based on the user's dMute balance (delegation balance to be more accurate) - the more dMute the user holds the higher APY they get.

However, the contract only checks the user's dMute balance at staking, the user doesn't have to hold that balance at any time but at the staking block.

This let's the user bribe somebody to delegate them their dMute for a single block and stake at the same time.

Since the balance checked is the delegation balance (`getPriorVotes`) this even makes it easier - since the 'lender' doesn't even need to trust the 'borrower' to return the funds, all the 'lender' can cancel the delegation on their own on the next block.

The lender can also create a smart contract to automate and decentralize this 'service'.

Impact

Users can get a share of the rewards that are supposed to incentivize them to hold dMute without actually holding them.

In other words, the same dMute tokens can be used to increase simultaneous staking of different users.

Proof of Concept

The following code shows that only a single block is being checked to calculate the `accountDTokenValue` at `calculateMultiplier()` :

```
        if(staked_block != 0 && enforce)
          accountDTokenValue = IDMute(dToken).getPriorVotes(acco
        else
          accountDTokenValue = IDMute(dToken).getPriorVotes(acco
```

The block checked when rewarding is the staked block, since `enforce` is true when applying reward:

```
            reward = lpTokenOut.mul(_totalWeight.sub(_userWeighted[a
```

Recommended Mitigation Steps

Make sure that the user holds the dMute for a longer time, one way to do it might be to sample a few random blocks between staking and current blocks and use the minimum balance as the user's balance.

mattt21 (Mute Switch) disputed and commented:

> dMute is a soulbound token that cannot be transferred. You can only receive it via burning mute for dmute with a minimum timelock for 7 days. This issue does not work as you cannot borrow dmute. You can borrow mute, and burn it for dmute, but you will be holding those dmute tokens for at least 7 days until you can redeem them back to mute.

> Please look at how the dMute token functions.

0xA5DF (warden) commented:

> Indeed, it can't be transferred. However it can be delegated, and since the amplifier checks for the delegation balance I think this attack path is still valid.

Picodes (judge) commented:

> @mattt21 - I do agree with @0xA5DF on this one. Unless I am missing something there is no restriction on delegations here, so the attack would work.

[M-03] `MuteAmplifier.rescueTokens()` checks the wrong condition for `muteToken`

Submitted by hansfriese*, also found by* evan

There will be 2 impacts.

- The reward system would be broken as the rewards can be withdrawn before starting staking.
- Some rewards would be locked inside the contract forever as it doesn't check `totalReclaimed`

Proof of Concept

`rescueTokens()` checks the below condition to rescue `muteToken`.

```
    else if (tokenToRescue == muteToken) {
        if (totalStakers > 0) {
            require(amount <= IERC20(muteToken).balanceOf(address(th
                "MuteAmplifier::rescueTokens: that muteToken belongs
            );
        }
    }
```

But there are 2 problems.

1. Currently, it doesn't check anything when `totalStakers == 0`. So some parts(or 100%) of rewards can be withdrawn before the staking period. In this case, the reward system won't work properly due to the lack of rewards.
2. It checks the wrong condition when `totalStakers > 0` as well. As we can see [here](), some remaining rewards are tracked using `totalReclaimed` and transferred to treasury directly. So we should consider this amount as well.

Recommended Mitigation Steps

It should be modified like the below.

```
    else if (tokenToRescue == muteToken) {
        if (totalStakers > 0) { //should check totalReclaimed as wel
            require(amount <= IERC20(muteToken).balanceOf(address(th
                "MuteAmplifier::rescueTokens: that muteToken belongs
            );
        }
        else if(block.timestamp <= endTime) { //no stakers but staki
```

```
            require(amount <= IERC20(muteToken).balanceOf(address(th
                "MuteAmplifier::rescueTokens: that muteToken belongs
            );
        }
    }
```

[mattt21 (Mute Switch) confirmed](#)

[M-04] An edge case in amplifier allows user to stake after end time, causing reward to be locked in the contract

Submitted by [evan](#)*, also found by* [HollaDieWaldfee](#), [hansfriese](#)*, and* [chaduke](#)

Observe that if nobody has staked after the period has ended, it's still possible for a single user to stake even though the period has ended.
https://github.com/code-423n4/2023-03-mute/blob/main/contracts/amplifier/MuteAmplifier.sol#L208-L212

```
            if (firstStakeTime == 0) {
                firstStakeTime = block.timestamp;
            } else {
                require(block.timestamp < endTime, "MuteAmplifier::s
            }
```

The staker can't get any of the rewards because the update modifier won't drip the rewards (since _mostRecentValueCalcTime = firstStakeTime >= endTime).
https://github.com/code-423n4/2023-03-mute/blob/main/contracts/amplifier/MuteAmplifier.sol#L89-L95

```
            if (_mostRecentValueCalcTime == 0) {
                _mostRecentValueCalcTime = firstStakeTime;
            }

            uint256 totalCurrentStake = totalStake();

            if (totalCurrentStake > 0 && _mostRecentValueCalcTime <
                ...
            }
```

At the same time, the protocol can't withdraw the rewards with rescueToken either since there is a staker, and no reward has been claimed yet (so the following check fails).
https://github.com/code-423n4/2023-03-mute/blob/main/contracts/amplifier/MuteAmplifier.sol#L187

```
            else if (tokenToRescue == muteToken) {
                if (totalStakers > 0) {
                    require(amount <= IERC20(muteToken).balanceOf(ac
                        "MuteAmplifier::rescueTokens: that muteToker
                    );
                }
            }
```

Impact

Suppose the staking period ends and nobody has staked. The admin would like to withdraw the rewards. A malicious user can front-run the rescueTokens call with a call to stake to lock all the rewards inside the contract indefinitely.

Recommended Mitigation Steps

https://github.com/code-423n4/2023-03-mute/blob/main/contracts/amplifier/MuteAmplifier.sol#L208-L212

The require shouldn't be inside the else block.

matt21 (Mute Switch) confirmed

[M-05] MuteBond is susceptible to DOS

Submitted by evan, also found by HollaDieWaldfee (1, 2)

https://github.com/code-423n4/2023-03-mute/blob/main/contracts/bonds/MuteBond.sol#L179
https://github.com/code-423n4/2023-03-

[mute/blob/main/contracts/dao/dMute.sol#L75-L77](https://github.com/code-423n4/2023-03-mute/blob/main/contracts/dao/dMute.sol#L75-L77)
https://github.com/code-423n4/2023-03-mute/blob/main/contracts/dao/dMute.sol#L57

Observe that if timeToTokens is called with `_lock_time = 1 week`, `_amount < 52`, it will return 0.
https://github.com/code-423n4/2023-03-mute/blob/main/contracts/dao/dMute.sol#L57

```
    function timeToTokens(uint256 _amount, uint256 _lock_time) inter
            uint256 week_time = 1 weeks;
            uint256 max_lock = 52 weeks;

            require(_lock_time >= week_time, "dMute::Lock: INSUFFICI
            require(_lock_time <= max_lock, "dMute::Lock: INSUFFICIE

            // amount * % of time locked up from min to max
            uint256 base_tokens = _amount.mul(_lock_time.mul(10**18)
            // apply % min max bonus
            //uint256 boosted_tokens = base_tokens.mul(lockBonus(loc

            return base_tokens;
        }
```

This causes lockTo to revert.
https://github.com/code-423n4/2023-03-mute/blob/main/contracts/dao/dMute.sol#L75-L77

```
     function LockTo(uint256 _amount, uint256 _lock_time, address to
            require(IERC20(MuteToken).balanceOf(msg.sender) >= _amou

            //transfer tokens to this contract
            IERC20(MuteToken).transferFrom(msg.sender, address(this)

            // calculate dTokens to mint
            uint256 tokens_to_mint = timeToTokens(_amount, _lock_tim

            require(tokens_to_mint > 0, 'dMute::Lock: INSUFFICIENT_T

            _mint(to, tokens_to_mint);
            _userLocks[to].push(UserLockInfo(_amount, block.timestam

            emit LockEvent(to, _amount, tokens_to_mint, _lock_time);
        }
```

The deposit function of muteBond calls lockTo with `_amount = payout`.
https://github.com/code-423n4/2023-03-mute/blob/main/contracts/bonds/MuteBond.sol#L179

```
IDMute(dMuteToken).LockTo(payout, bond_time_lock, _depositor);
```

Observe that regardless what the inputs are, `payout <= maxDeposit()` is always satisfied after the following code segment.
https://github.com/code-423n4/2023-03-mute/blob/main/contracts/bonds/MuteBond.sol#L155-L164

```
            uint payout = payoutFor( value );
            if(max_buy == true){
              value = maxPurchaseAmount();
              payout = maxDeposit();
            } else {
              // safety checks for custom purchase
              require( payout >= ((10**18) / 100), "Bond too small"
              require( payout <= maxPayout, "Bond too large"); // si
              require( payout <= maxDeposit(), "Deposit too large");
            }
```

So, if an attacker manipulates the muteBond to get maxDeposit() < 52, deposit will always fail.

Please add the following test case to bonds.ts and run it with `npm run test-bond`

Note that if the bond price is too high (> 52e18), then this won't always be possible (because payout will change by bondPrice every time we increment/decrement value). So in my POC, I set the price range to be (1e18 - 2e18), which I believe are reasonable values as well.

```
    it('Bond DOS', async function () {
```

```
await bondContract.setStartPrice(new BigNumber(1).times(Math
await bondContract.setMaxPrice(new BigNumber(2).times(Math.p
await bondContract.setMaxPayout(new BigNumber(100).times(Mat

// ideally, the following line is what I had in mind
// var val = new BigNumber((await bondContract.maxPurchaseAm
// but due to timing issues I couldn't get it to work (I'm n

// so I just ran this to get the value for the next line
// await time.increase(1)
// console.log(await bondContract.maxPurchaseAmount());


var val = new BigNumber("9999851192690584965З").toFixed();

console.log("before:")
console.log(await bondContract.maxDeposit())
/*
console.log(await bondContract.payoutFor(val))
console.log(await bondContract.maxPayout())
console.log((await bondContract.maxPurchaseAmount()))
*/

await bondContract.connect(buyer1).deposit(val, buyer1.addre

console.log("after:")
console.log(await bondContract.maxDeposit())
/*
console.log(await bondContract.payoutFor(val))
console.log(await bondContract.maxPayout())
console.log((await bondContract.maxPurchaseAmount()))
*/

await expect(
  bondContract.connect(buyer1).deposit(1, buyer1.address, fa
).to.be.reverted;


await expect(
  bondContract.connect(buyer1).deposit(1, buyer1.address, tr
).to.be.reverted;


await expect(
  bondContract.connect(buyer1).deposit(new BigNumber(1).time
).to.be.reverted;
```

```
        await expect(
          bondContract.connect(buyer1).deposit(new BigNumber(1).time
        ).to.be.reverted;
      })
```

Impact

This vulnerability causes deposit to fail indefinitely. That being said, the contract itself doesn't seem to store funds, and it looks like there are ways for the admin to manually fix the DOS (e.g. deploy a new contract, set startPrice / maxPrice). So overall, I would say it warrants a medium severity.

Tools Used

Manual Review, Hardhat

Recommended Mitigation Steps

Start a new epoch if maxDeposit() is smaller than a certain threshold.

[mattt21 (Mute Switch) confirmed]

[M-06] Amplifier users might not get all the LP fees they are entitled to

Submitted by [evan]

Observe that there is only one place that the amplifier is calling claimFees, and it's inside an if statement of the update modifier, requiring `_mostRecentValueCalcTime < endTime`.

[https://github.com/code-423n4/2023-03-mute/blob/main/contracts/amplifier/MuteAmplifier.sol#L111]

```
    modifier update() {
            if (_mostRecentValueCalcTime == 0) {
                _mostRecentValueCalcTime = firstStakeTime;
```

```
        }

        uint256 totalCurrentStake = totalStake();

        if (totalCurrentStake > 0 && _mostRecentValueCalcTime <
            uint256 value = 0;
            uint256 sinceLastCalc = block.timestamp.sub(_mostRec
            uint256 perSecondReward = totalRewards.div(endTime.s

            if (block.timestamp < endTime) {
                value = sinceLastCalc.mul(perSecondReward);
            } else {
                uint256 sinceEndTime = block.timestamp.sub(endTi
                value = (sinceLastCalc.sub(sinceEndTime)).mul(pe
            }

            _totalWeight = _totalWeight.add(value.mul(10**18).di

            _mostRecentValueCalcTime = block.timestamp;

            (uint fee0, uint fee1) = IMuteSwitchPairDynamic(lpTo

            _totalWeightFee0 = _totalWeightFee0.add(fee0.mul(10*
            _totalWeightFee1 = _totalWeightFee1.add(fee1.mul(10*

            totalFees0 = totalFees0.add(fee0);
            totalFees1 = totalFees1.add(fee1);
        }

        _;
    }
```

Consider the following situation. An user X has staked a large amount of LP tokens, and a user Y has staked a normal amount.

Y withdraws as soon as the staking period ends (block.timestamp > endTime), triggering the update modifier, which sets `_mostRecentValueCalcTime = block.timestamp` > endTime. Observe that after this point, the amplifier will never call claimFees again since `_mostRecentValueCalcTime < endTime` will forever be false.

Meanwhile, X forgot about it, and doesn't withdraw until say 2 weeks after endTime. When X calls withdraw, X won't get the LP fees for those 2 weeks. In fact, nobody will - they are trapped inside the mute switch pair forever since the amplifier won't call claim.

Impact

Some LP fees can be trapped inside the mute switch pair when it should really be going to the amplifier users.

Recommended Mitigation Steps

I believe it's best to move the LP fee calculation out of the if statement.

mattt21 (Mute Switch) confirmed

[M-07] `MuteAmplifier.sol` : multiplier calculation is incorrect which leads to loss of rewards for almost all stakers

Submitted by HollaDieWaldfee

https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/amplifier/MuteAmplifier.sol#L473-L499
https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/amplifier/MuteAmplifier.sol#L366-L388
https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/amplifier/MuteAmplifier.sol#L417-L460

This report deals with how the calculation of the `multiplier` in the `MuteAmplifier` contract is not only different from how it is displayed in the documentation on the website but it is also different in a very important way.

The calculation on the website shows a linear relationship between the `dMUTE / poolSize` ratio and the `APY`. The `dMUTE / poolSize` ratio is also called the `tokenRatio`.

By “linear” I mean that when a user increases his `tokenRatio` from `0` to `0.1` this has the same effect as when increasing it from `0.9` to `1`.

The implementation in the `MuteAmplifier.calculateMultiplier` function does not have this linear relationship between `tokenRatio` and `APY`.

An increase in the `tokenRatio` from `0` to `0.1` is worth much less than an increase from `0.9` to `1`.

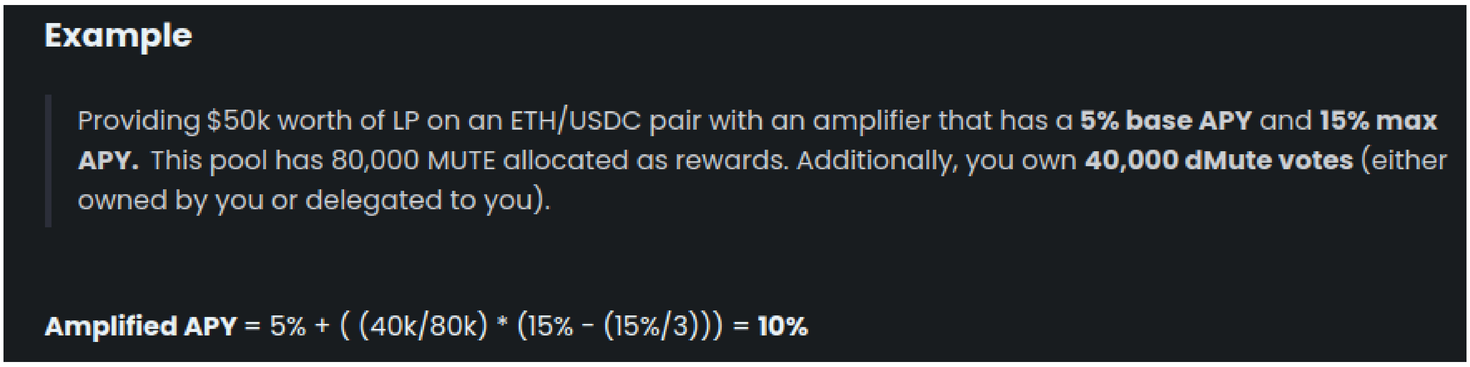
As we will see this means that all stakers that do not have a `tokenRatio` of exactly equal `0` or exactly equal `1` lose out on rewards that they should receive according to the documentation.

I estimate this to be of “High” severity because the issue affects nearly all stakers and results in a partial loss of rewards.

Proof of Concept

Let’s first look at the `multiplier` calculation from the documentation:

$$f(APY_{amp}) = APY_{base} + (clamp(user_{dmute}/pool_{rewards}, 1) * (APY_{max} - (APY_{max}/3)))$$



Example

> Providing $50k worth of LP on an ETH/USDC pair with an amplifier that has a **5% base APY** and **15% max APY.** This pool has 80,000 MUTE allocated as rewards. Additionally, you own **40,000 dMute votes** (either owned by you or delegated to you).

Amplified APY = 5% + ((40k/80k) * (15% - (15%/3))) = **10%**

The example calculation shows that the amount that is added to $APY\_{base}$ (5%) is scaled linearly by the $\dfrac{user\_{dmute}}{pool\_{rewards}}$ ratio which I called `tokenRatio` above.

This means that when a user increases his `tokenRatio` from say `0` to `0.1` he gets the same additional reward as when he increases the `tokenRatio` from say `0.9` to `1`.

Let's now look at how the reward and thereby the `multiplier` is calculated in the code.

The first step is to calculate the `multiplier` which happens in the `MuteAmplifier.calculateMultiplier` function:

Link

```
    function calculateMultiplier(address account, bool enforce) publ
        require(account != address(0), "MuteAmplifier::calculateMult

        uint256 accountDTokenValue;

        // zkSync block.number = L1 batch number. This at times is t
        // we take the previous block into account
        uint256 staked_block =  _userStakedBlock[account] == block.r

        if(staked_block != 0 && enforce)
            accountDTokenValue = IDMute(dToken).getPriorVotes(accour
        else
            accountDTokenValue = IDMute(dToken).getPriorVotes(accour

        if(accountDTokenValue == 0){
            return _stakeDivisor;
        }

        uint256 stakeDifference = _stakeDivisor.sub(10 ** 18);

        // ratio of dMute holdings to pool
        uint256 tokenRatio = accountDTokenValue.mul(10**18).div(tota

        stakeDifference = stakeDifference.mul(clamp_value(tokenRatic

        return _stakeDivisor.sub(stakeDifference);
    }
```

The `multiplier` that is returned is then used to calculate the `reward`:

Link

```
    reward = lpTokenOut.mul(_totalWeight.sub(_userWeighted[account])
```

Let’s write the formula in a more readable form:

$\dfrac{lpTokenOut * weightDifference}{stakeDivisor - tokenRatio * (stakeDivisor - 1)}$

$stakeDivisor$ can be any number $>=1$ and has the purpose of determining the percentage of rewards a user with $tokenRatio=0$ gets.

For the sake of this argument we can assume that all numbers except $tokenRatio$ are constant.

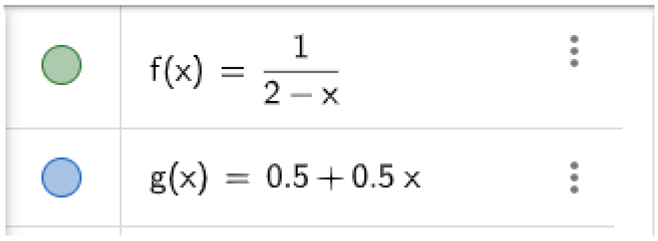
Let’s just say $stakeDivisor=2$ which means that a user with `$tokenRatio=0$` would receive $\dfrac{1}{2}=50%$ of the maximum rewards.

Further let’s say that $lpTokenOut * weightDifference = 1$, so 100% of the possible rewards would be 1.

We can then write the formula like this:

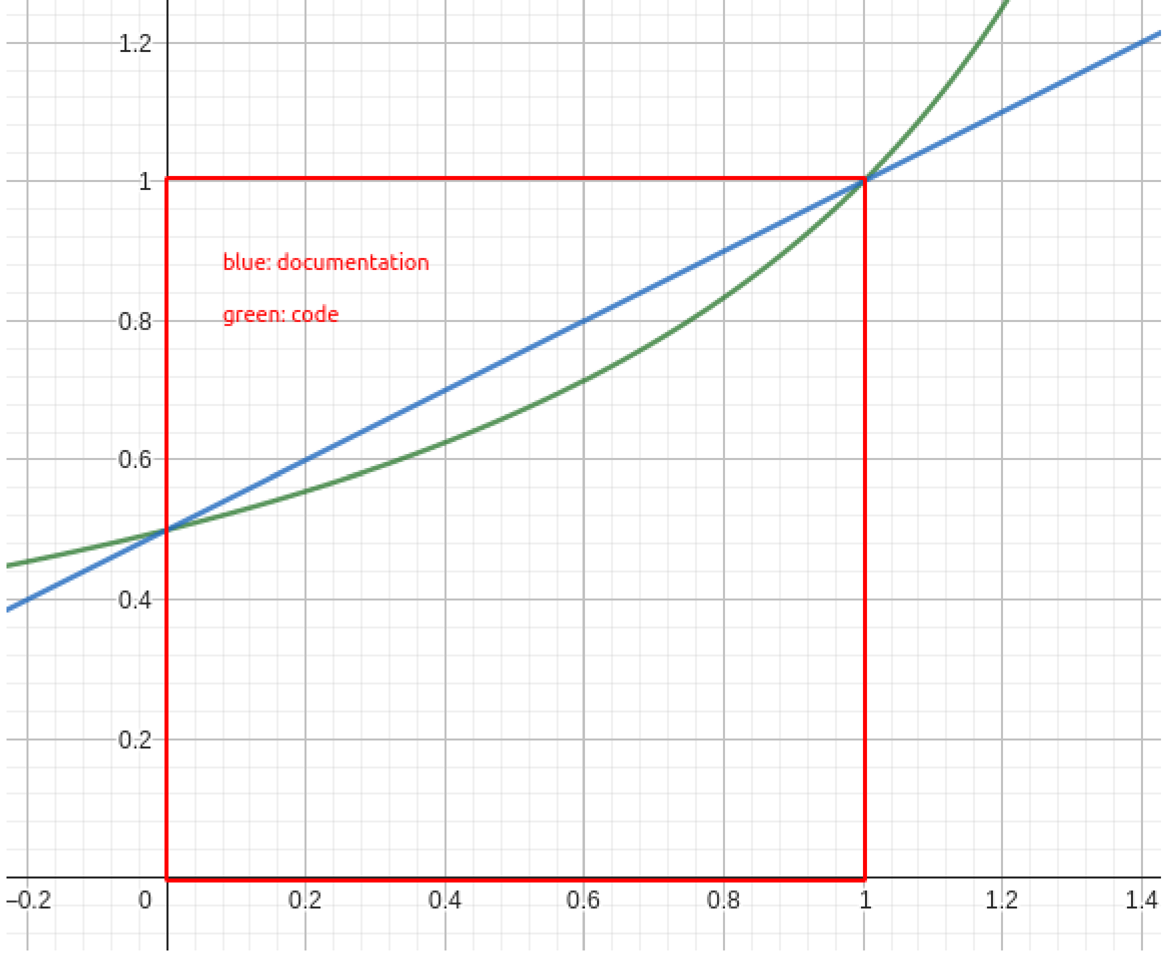
$\dfrac{1}{2 - tokenRatio}$

So let’s compare the calculation from the documentation with the calculation from the code by looking at a plot:



🟢 $f(x) = \dfrac{1}{2 - x}$

🔵 $g(x) = 0.5 + 0.5\,x$



x-axis: tokenRatio
y-axis: percentage of maximum rewards

We can see that the green curve is non-linear and below the blue curve.

So the rewards as calculated in the code are too low.

Tools Used

VSCode

Recommended Mitigation Steps

I recommend to change the reward calculation to this:

$(lpTokenOut * weightDifference) * (percentage\_{min} + clamp(\dfrac{user\_{dmute}}{pool\_{rewards}},1) * (1 - percentage\_{min}))$

Instead of setting the `stakeDivisor` upon initialization, the `percentageMin` should be set which can be in the interval `[0,1e18]`.

Fix:

```
diff --git a/contracts/amplifier/MuteAmplifier.sol b/contracts/a
index 9c6fcb5..1c86f5c 100644
--- a/contracts/amplifier/MuteAmplifier.sol
+++ b/contracts/amplifier/MuteAmplifier.sol
@@ -48,7 +48,7 @@ contract MuteAmplifier is Ownable{

     uint256 private _mostRecentValueCalcTime; // latest update

-    uint256 public _stakeDivisor; // divisor set in place for m
+    uint256 public _percentageMin; // minimum percentage set ir

     uint256 public management_fee; // lp withdrawal fee
     address public treasury; // address that receives the lp wi
@@ -131,8 +131,8 @@ contract MuteAmplifier is Ownable{
      *  @param _mgmt_fee uint
      *  @param _treasury address
      */
-    constructor (address _lpToken, address _muteToken, address
-        require(divisor >= 10 ** 18, "MuteAmplifier: invalid _s
+    constructor (address _lpToken, address _muteToken, address
+        require(_percentageMin <= 10 ** 18, "MuteAmplifier: inv
         require(_lpToken != address(0), "MuteAmplifier: invalic
         require(_muteToken != address(0), "MuteAmplifier: inval
         require(_dToken != address(0), "MuteAmplifier: invalid
@@ -142,7 +142,7 @@ contract MuteAmplifier is Ownable{
         lpToken = _lpToken;
         muteToken = _muteToken;
         dToken = _dToken;
-        _stakeDivisor = divisor;
+        _percentageMin = percentageMin;
         management_fee = _mgmt_fee; //bps 10k
         treasury = _treasury;

@@ -368,7 +368,7 @@ contract MuteAmplifier is Ownable{
         require(lpTokenOut > 0, "MuteAmplifier::_applyReward: r

         // current rewards based on multiplier
-        reward = lpTokenOut.mul(_totalWeight.sub(_userWeighted[
+        reward = lpTokenOut.mul(_totalWeight.sub(_userWeighted[
         // max possible rewards
         remainder = lpTokenOut.mul(_totalWeight.sub(_userWeight
         // calculate left over rewards
@@ -442,7 +442,7 @@ contract MuteAmplifier is Ownable{
             uint256 _totalWeightFee1Local = _totalWeightFee1.ac
```

```
                // current rewards based on multiplier
-               info.currentReward = totalUserStake(user).mul(totWe
+               info.currentReward = totalUserStake(user).mul(totWe
                // add back any accumulated rewards
                info.currentReward = info.currentReward.add(_userAc

@@ -452,7 +452,7 @@ contract MuteAmplifier is Ownable{

            } else {
              // current rewards based on multiplier
-             info.currentReward = totalUserStake(user).mul(_totalW
+             info.currentReward = totalUserStake(user).mul(_totalW
              // add back any accumulated rewards
              info.currentReward = info.currentReward.add(_userAccu
            }
@@ -485,17 +485,17 @@ contract MuteAmplifier is Ownable{
              accountDTokenValue = IDMute(dToken).getPriorVotes(acc

            if(accountDTokenValue == 0){
-             return _stakeDivisor;
+             return _percentageMin;
            }

-           uint256 stakeDifference = _stakeDivisor.sub(10 ** 18);
+           uint256 percentageDifference = (uint256(10 ** 18)).sub

            // ratio of dMute holdings to pool
            uint256 tokenRatio = accountDTokenValue.mul(10**18).div

-           stakeDifference = stakeDifference.mul(clamp_value(toker
+           uint256 additionalPercentage = percentageDifference.mul

-           return _stakeDivisor.sub(stakeDifference);
+           return _percentageMin.add(additionalPercentage);
        }
```

mattt21 (Mute Switch) confirmed

Picodes (judge) decreased severity to Medium and commented:

> This is a remarkable finding. However, I'll downgrade it to medium as assets are not strictly speaking directly at risk (they can't be stolen and the state of the system cannot be manipulated to grieve another user).

> We could also argue that this is a case of "function incorrect as to spec" which is according to C4 doc of low severity.

> Considering the importance of the finding, and especially the fact that it could lead to users receiving fewer rewards than they expected, I think Medium severity is appropriate.

[M-08] `MuteAmplifier.sol`: `rescueTokens` function does not prevent fee tokens from being transferred

Submitted by HollaDieWaldfee, also found by evan and hansfriese

The `MuteAmplifier.rescueTokens` function allows the `owner` to withdraw tokens that are not meant to be in this contract.

The contract does protect tokens that ARE meant to be in the contract by not allowing them to be transferred:

Link

```
function rescueTokens(address tokenToRescue, address to, uint256
    if (tokenToRescue == lpToken) {
        require(amount <= IERC20(lpToken).balanceOf(address(this
            "MuteAmplifier::rescueTokens: that Token-Eth belongs
        );
    } else if (tokenToRescue == muteToken) {
        if (totalStakers > 0) {
            require(amount <= IERC20(muteToken).balanceOf(addres
                "MuteAmplifier::rescueTokens: that muteToken bel
            );
        }
    }


    IERC20(tokenToRescue).transfer(to, amount);
}
```

You can see that `lpToken` and `muteToken` cannot be transferred unless there is an excess amount beyond what is needed by the contract.

So stakers can be sure that not even the contract `owner` can mess with their stakes.

The issue is that `lpToken` and `muteToken` are not the only tokens that need to stay in the contract.

There is also the `fee0` token and the `fee1` token.

So what can happen is that the `owner` can withdraw `fee0` or `fee1` tokens and users cannot `payout` rewards or `withdraw` their stake because the transfer of `fee0` / `fee1` tokens reverts due to the missing balance.

Users can of course send `fee0` / `fee1` tokens to the contract to restore the balance. But this is not intended and certainly leaves the user worse off.

Proof of Concept

Assume that when an update occurs via the `update` modifier there is an amount of `fee0` tokens claimed:

Link

```
    modifier update() {
        if (_mostRecentValueCalcTime == 0) {
            _mostRecentValueCalcTime = firstStakeTime;
        }


        uint256 totalCurrentStake = totalStake();


        if (totalCurrentStake > 0 && _mostRecentValueCalcTime <
            uint256 value = 0;
            uint256 sinceLastCalc = block.timestamp.sub(_mostRec
            uint256 perSecondReward = totalRewards.div(endTime.s


            if (block.timestamp < endTime) {
                value = sinceLastCalc.mul(perSecondReward);
            } else {
                uint256 sinceEndTime = block.timestamp.sub(endTi
                value = (sinceLastCalc.sub(sinceEndTime)).mul(pe
            }

            _totalWeight = _totalWeight.add(value.mul(10**18).di

            _mostRecentValueCalcTime = block.timestamp;

            (uint fee0, uint fee1) = IMuteSwitchPairDynamic(lpTo

            _totalWeightFee0 = _totalWeightFee0.add(fee0.mul(10*
            _totalWeightFee1 = _totalWeightFee1.add(fee1.mul(10*

            totalFees0 = totalFees0.add(fee0);
            totalFees1 = totalFees1.add(fee1);
        }

        _;
    }
```

We can see that `_totalWeightFee0` is updated such that when a user's rewards are calculated the `fee0` tokens will be paid out to the user.

What happens now is that the `owner` calls `rescueTokens` which transfers the `fee0` tokens out of the contract.

We can see that when the `fee0` to be paid out to the user is calculated in the `_applyReward` function, the calculation is solely based on `_totalWeightFee0` and does not take into account if the `fee0` tokens still exist in the contract.

Link

```
fee0 = lpTokenOut.mul(_totalWeightFee0.sub(_userWeightedFee0[acc
```

So when the `fee0` tokens are attempted to be transferred to the user that calls `payout` or `withdraw`, the transfer reverts due to insufficient balance.

Tools Used

VSCode

Recommended Mitigation Steps

The `MuteAmplifier.rescueTokens` function should check that only excess `fee0` / `fee1` tokens can be paid out. Such that tokens that will be paid out to stakers need to stay in the contract.

Fix:

```
diff --git a/contracts/amplifier/MuteAmplifier.sol b/contracts/a
index 9c6fcb5..b154d81 100644
--- a/contracts/amplifier/MuteAmplifier.sol
+++ b/contracts/amplifier/MuteAmplifier.sol
@@ -188,6 +188,18 @@ contract MuteAmplifier is Ownable{
                     "MuteAmplifier::rescueTokens: that muteToke
                 );
             }
+        } else if (tokenToRescue == address(IMuteSwitchPairDyna
+            if (totalStakers > 0) {
+                require(amount <= IERC20(IMuteSwitchPairDynamic
+                    "MuteAmplifier::rescueTokens: that token be
+                );
+            }
+        } else if (tokenToRescue == address(IMuteSwitchPairDyna
+            if (totalStakers > 0) {
+                require(amount <= IERC20(IMuteSwitchPairDynamic
+                    "MuteAmplifier::rescueTokens: that token be
+                );
+            }
         }
 
         IERC20(tokenToRescue).transfer(to, amount);
```

The issue discussed in this report also ties in with the fact that the `fee0 <= totalFees0 && fee1 <= totalFees1` check before transferring fee tokens always passes. It does not prevent the scenario that the sponsor wants to prevent which is when there are not enough fee tokens to be transferred the transfer should not block the function.

So in addition to the above changes I propose to add these changes as well:

```
diff --git a/contracts/amplifier/MuteAmplifier.sol b/contracts/a
index 9c6fcb5..39cd75b 100644
--- a/contracts/amplifier/MuteAmplifier.sol
+++ b/contracts/amplifier/MuteAmplifier.sol
@@ -255,7 +255,7 @@ contract MuteAmplifier is Ownable{
         }

         // payout fee0 fee1
-        if ((fee0 > 0 || fee1 > 0) && fee0 <= totalFees0 && fee
+        if ((fee0 > 0 || fee1 > 0) && fee0 < IERC20(IMuteSwitch
             address(IMuteSwitchPairDynamic(lpToken).token0()).s
             address(IMuteSwitchPairDynamic(lpToken).token1()).s

@@ -295,7 +295,7 @@ contract MuteAmplifier is Ownable{
         }

         // payout fee0 fee1
-        if ((fee0 > 0 || fee1 > 0) && fee0 <= totalFees0 && fee
+        if ((fee0 > 0 || fee1 > 0) && fee0 < IERC20(IMuteSwitch
             address(IMuteSwitchPairDynamic(lpToken).token0()).s
             address(IMuteSwitchPairDynamic(lpToken).token1()).s

@@ -331,7 +331,7 @@ contract MuteAmplifier is Ownable{
             }

             // payout fee0 fee1
-            if ((fee0 > 0 || fee1 > 0) && fee0 <= totalFees0 &&
+            if ((fee0 > 0 || fee1 > 0) && fee0 < IERC20(IMuteSw
                 address(IMuteSwitchPairDynamic(lpToken).token0(
                 address(IMuteSwitchPairDynamic(lpToken).token1(
```

mattt21 (Mute Switch) confirmed

[M-09] `MuteBond.sol`: When `maxPayout` is lowered the contract can end up DOSed

Submitted by HollaDieWaldfee*, also found by* hansfriese*,* 0xA5DF*, and* chaduke

https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/bonds/MuteBond.sol#L119-L123
https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/bonds/MuteBond.sol#L153-L200

The `maxPayout` variable in the `MuteBond` contract specifies the amount of `MUTE` that is paid out in one epoch before the next epoch is entered.

The variable is initialized in the constructor and can then be changed via the `setMaxPayout` function.

The issue occurs when `maxPayout` is lowered.

So say `maxPayout` is currently `10,000 MUTE` and the `owner` wants to reduce it to `5,000 MUTE`.

Before this transaction to lower `maxPayout` is executed, another transaction might be executed which increases the current payout to `> 5,000 MUTE`.

This means that calls to `MuteBond.deposit` revert and no new epoch can be entered. Thereby the `MuteBond` contracts becomes unable to provide bonds.

The DOS is not permanent. The `owner` can increase `maxPayout` such that the current payout is smaller than `maxPayout` again and the contract will work as intended.

So the impact is a temporary DOS of the `MuteBond` contract.

The issue can be solved by requiring in the `setMaxPayout` function that `maxPayout` must be bigger than the payout in the current epoch.

Proof of Concept

Add the following test to the `bonds.ts` test file:

```
it('POC maxPayout below current payout causes DOS', async functi
    // owner wants to set maxPayout to 9 * 10**18
    // However a transaction is executed first that puts the pay
    // all further deposits revert

    // make a payout
    await bondContract.connect(owner).deposit(new BigNumber(10).

    // set maxPayout below currentPayout
    await bondContract.connect(owner).setMaxPayout(new BigNumber

    // deposit reverts due to underflow
    await bondContract.connect(owner).deposit("0", owner.address
  })
```

Tools Used

VSCode

Recommended Mitigation Steps

I recommend that the `setMaxPayout` function checks that `maxPayout` is set to a value bigger than the payout in the current epoch:

```
diff --git a/contracts/bonds/MuteBond.sol b/contracts/bonds/Mute
index 96ee755..4af01d7 100644
--- a/contracts/bonds/MuteBond.sol
+++ b/contracts/bonds/MuteBond.sol
@@ -118,6 +118,7 @@ contract MuteBond {
      */
     function setMaxPayout(uint _payout) external {
         require(msg.sender == customTreasury.owner());
+        require(_payout > terms[epoch].payoutTotal, "_payout to
         maxPayout = _payout;
         emit MaxPayoutChanged(_payout);
     }
```

mattt21 (Mute Switch) confirmed via duplicate issue #35

Low Risk and Non-Critical Issues

For this audit, 5 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **HollaDieWaldfee** received the top score from the judge.

The following wardens also submitted reports: evan, 0xA5DF, hansfriese, and chaduke.

Summary

Risk	Title	File	Instances
L-01	Use fixed compiler version	-	3
L-02	`RedeemEvent` emits wrong `to` address	dMute.sol	1
L-03	Replicate price checks from constructor in setter functions	MuteBond.sol	2
L-04	Ownable: Does not implement 2-Step-Process for transferring ownership	MuteAmplifier.sol	1
L-05	Only allow rescuing `MUTE` rewards when `endTime` is reached	MuteAmplifier.sol	1
L-06	First user that stakes again after a period without stakers receives too many rewards	MuteAmplifier.sol	1
L-07	`dripInfo` function reverts when `firstStakeTime >= endTime`	MuteAmplifier.sol	1
L-08	`dripInfo` function does not calculate `fee0` and `fee1` in the `else` block	MuteAmplifier.sol	1
L-09	Check for each fee token individually that non-zero value is transferred	MuteAmplifier.sol	1
N-01	Remove `require` statements that are always true	dMute.sol	2
N-02	Remove `SafeMath` library	-	3

Risk	Title	File	Instances
N-03	Event parameter names are messed up	MuteBond.sol	-
N-04	Event is never emitted	-	2
N-05	Move `payoutFor` calculation into `else` block	MuteBond.sol	1
N-06	Remove redundant check in `stake` function	MuteAmplifier.sol	1

[L-01] Use fixed compiler version

All in scope contracts use `^0.8.0` as compiler version.

They should use a fixed version, i.e. `0.8.12`, to make sure the contracts are always compiled with the intended version.

https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/amplifier/MuteAmplifier.sol#L2

https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/bonds/MuteBond.sol#L2

https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/dao/dMute.sol#L2

[L-02] `RedeemEvent` emits wrong `to` address

In the `dMute.RedeemTo` function the `RedeemEvent` is emitted (Link).

It is defined as:
Link

```
event RedeemEvent(address to, uint256 unlockedAmount, uint256 bu
```

So the first parameter should be the `to` address.

When the event is emitted the first parameter is `msg.sender`. The issue is that the `to` address and `msg.sender` can be different. So the event in some cases contains wrong information.

Fix:

```
diff --git a/contracts/dao/dMute.sol b/contracts/dao/dMute.sol
index 59f95b7..98a65d5 100644
--- a/contracts/dao/dMute.sol
+++ b/contracts/dao/dMute.sol
@@ -125,7 +125,7 @@ contract dMute is dSoulBound {
         _burn(msg.sender, total_to_burn);


-        emit RedeemEvent(msg.sender, total_to_redeem, total_to_
+        emit RedeemEvent(to, total_to_redeem, total_to_burn);
     }

     function GetUserLockLength(address account) public view ret
```

[L-03] Replicate price checks from constructor in setter functions

In the `MuteBond` constructor it is checked that `maxPrice >= startPrice` ([Link]()).

These checks are not implemented in the [`setStartPrice`]() and [`setMaxPrice`]() setter functions.

It is recommended to add the check to both setter functions such that it is ensured the setter functions do not cause `startPrice` and `maxPrice` to be set to bad values.

Fix:

```
diff --git a/contracts/bonds/MuteBond.sol b/contracts/bonds/Mute
index 96ee755..49b87f9 100644
--- a/contracts/bonds/MuteBond.sol
+++ b/contracts/bonds/MuteBond.sol
@@ -98,6 +98,7 @@ contract MuteBond {
      */
     function setMaxPrice(uint _price) external {
         require(msg.sender == customTreasury.owner());
+        require(_price >= startPrice, "starting price < min");
         maxPrice = _price;
         emit MaxPriceChanged(_price);
     }
@@ -108,6 +109,7 @@ contract MuteBond {
      */
     function setStartPrice(uint _price) external {
         require(msg.sender == customTreasury.owner());
+        require(maxPrice >= _price, "starting price < min");
         startPrice = _price;
         emit StartPriceChanged(_price);
     }
```

[L-04] Ownable: Does not implement 2-Step-Process for transferring ownership

`MuteAmplifier` inherits from the `Ownable` contract.
This contract does not implement a `2-Step-Process` for transferring ownership.
So ownership of the contract can easily be lost when making a mistake when transferring ownership.

Consider using the `Ownable2Step` contract from OZ (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable2Step.sol) instead.

[L-05] Only allow rescuing `MUTE` rewards when `endTime` is reached

The `MuteAmplifier.rescueTokens` function allows the `owner` to rescue any tokens from the contract.
In the case of `MUTE` it is checked if `totalStakers > 0`, i.e. if there are any stakers.
If there are stakers then only tokens in excess of rewards can be rescued.

If there are no stakers, all `MUTE` tokens can be rescued.
I argue that this behavior does not what is intended. The issue is that there might just temporarily be no stakers but the `endTime` is not reached yet. This means the contract should be able to payout rewards.

A user that stakes when there are no `MUTE` rewards (there must still be a small excess balance of `MUTE` , e.g. sent by an attacker with griefing intent, to pass this check in the stake function) must send `MUTE` to the contract in order to be able to `withdraw` again. Otherwise an amount of `MUTE` is attempted to be transferred that is not held in the contract.

Based on the limited privileges the `owner` has I don't think the behavior described above is what is intended.

So I recommend that the `rescueTokens` function should not allow rescuing `MUTE` rewards within the `startTime` and `endTime` at all.

Fix:

```
diff --git a/contracts/amplifier/MuteAmplifier.sol b/contracts/a
index 9c6fcb5..55ee81b 100644
--- a/contracts/amplifier/MuteAmplifier.sol
+++ b/contracts/amplifier/MuteAmplifier.sol
@@ -183,7 +183,7 @@ contract MuteAmplifier is Ownable{
                 "MuteAmplifier::rescueTokens: that Token-Eth be
             );
         } else if (tokenToRescue == muteToken) {
-            if (totalStakers > 0) {
+            if (block.timestamp >= startTime && startTime !=0 &
                 require(amount <= IERC20(muteToken).balanceOf(a
                     "MuteAmplifier::rescueTokens: that muteToke
                 );
```

Note: I submitted a similar report that deals with rescuing fee tokens as "Medium" severity. I did this because in the case of rescuing fee tokens it affects EXISTING stakers. Here it affects only stakers that stake AFTER the tokens have been rescued.

[L-06] First user that stakes again after a period without stakers receives too many rewards

The `MuteAmplifier` contract pays out rewards on a per second basis.

Let's assume there is only 1 staker which is Bob.

Say Bob calls `stake` at `timestamp 0` and calls `withdraw` at `timestamp 10`. He receives rewards for 10 seconds of staking.

At `timestsamp 30` Bob calls `stake` again (there were no stakers from `timestamp 10` to `timestamp 30`).

If Bob calls `withdraw` at say `timestamp 40`, he receives not only rewards for the 10 seconds he has staked but for 30 seconds (`timestamp 10` to `timestamp 40`).

This means that whenever there are temporarily no stakers, whoever first stakes again receives all the rewards from the previous period without stakers.

This is due to how the `update` modifier works.

When someone stakes and there were no other stakers, the `if` block is not entered and the `_mostRecentValueCalcTime` variable is not updated.

So when the `update` modifier is executed again the staker also receives the rewards from the period when there were no stakers.

I just want to make the sponsor aware of this behavior. The sponsor may decide that this is unintended and needs to change. I think this might even be a beneficial behavior because it incentivises users to stake if there are no stakers because they will get more rewards.

[L-07] `dripInfo` function reverts when `firstStakeTime >= endTime`

It is unlikely but possible that `firstStakeTime >= endTime`.

I suggest in issue 47 that staking should only occur when `block.timestamp < endTime` which would mitigate this issue as well. But for now this issue exists.

So when `firstStakeTime >= endTime`, the following line in the `dripInfo` function reverts:

Link

```
info.perSecondReward = totalRewards.div(endTime.sub(firstStakeTi
```

The current behavior can cause DOS issues in any components that make use of this function.

I recommend to either implement the changes in `[L-05]` or to implement the following:

```
diff --git a/contracts/amplifier/MuteAmplifier.sol b/contracts/a
index 9c6fcb5..415da7f 100644
--- a/contracts/amplifier/MuteAmplifier.sol
+++ b/contracts/amplifier/MuteAmplifier.sol
@@ -416,7 +416,11 @@ contract MuteAmplifier is Ownable{
      */
     function dripInfo(address user) external view returns (Drip

-        info.perSecondReward = totalRewards.div(endTime.sub(fir
+        if (endTime > firstStakeTime) {
+            info.perSecondReward = totalRewards.div(endTime.sub
+        } else {
+            info.perSecondReward = 0;
+        }
         info.totalLP = _totalStakeLpToken;
         info.multiplier_current = calculateMultiplier(user, fal
         info.multiplier_last = calculateMultiplier(user, true);
```

[L-08] `dripInfo` function does not calculate `fee0` and `fee1` in the `else` block

In the `else` block, the `MuteAmplifier.dripInfo` function does not calculate the value for `info.fee0` and `info.fee1`.

Fix:

```
diff --git a/contracts/amplifier/MuteAmplifier.sol b/contracts/a
index 9c6fcb5..20974b8 100644
--- a/contracts/amplifier/MuteAmplifier.sol
+++ b/contracts/amplifier/MuteAmplifier.sol
@@ -455,6 +455,9 @@ contract MuteAmplifier is Ownable{
           info.currentReward = totalUserStake(user).mul(_totalW
           // add back any accumulated rewards
           info.currentReward = info.currentReward.add(_userAccu
+
+          info.fee0 = totalUserStake(user).mul(_totalWeightFee(
+          info.fee1 = totalUserStake(user).mul(_totalWeightFee1
         }

     }
```

[L-09] Check for each fee token individually that non-zero value is transferred

The `MuteAmplifier` contract executes the following code when fee tokens are transferred:

Link

```
if ((fee0 > 0 || fee1 > 0) && fee0 <= totalFees0 && fee1 <= tota
    address(IMuteSwitchPairDynamic(lpToken).token0()).safeTransf
    address(IMuteSwitchPairDynamic(lpToken).token1()).safeTransf
```

So when one of the fee tokens has a value that is greater 0, both fee tokens are transferred. This means that it is possible for a transfer to occur with the zero value.

There exist tokens that revert on zero-value transfer. Also the upstream contract that transfers the fee tokens to the `MuteAmplifier` contract checks for each token individually that the amount is greater zero:

Link

```
function claimFeesFor(address recipient, uint amount0, uint amou
    require(msg.sender == pair);
    if (amount0 > 0) _safeTransfer(token0, recipient, amount0);
    if (amount1 > 0) _safeTransfer(token1, recipient, amount1);
}
```

Therefore I recommend to check in the `MuteAmplifier` contract the value for each token individually as well.

Fix:

```
diff --git a/contracts/amplifier/MuteAmplifier.sol b/contracts/a
index 9c6fcb5..42b4ec2 100644
--- a/contracts/amplifier/MuteAmplifier.sol
+++ b/contracts/amplifier/MuteAmplifier.sol
@@ -256,8 +256,8 @@ contract MuteAmplifier is Ownable{

         // payout fee0 fee1
         if ((fee0 > 0 || fee1 > 0) && fee0 <= totalFees0 && fee
-            address(IMuteSwitchPairDynamic(lpToken).token0()).s
-            address(IMuteSwitchPairDynamic(lpToken).token1()).s
+            if (fee0 > 0) address(IMuteSwitchPairDynamic(lpToke
+            if (fee1 > 0) address(IMuteSwitchPairDynamic(lpToke

             totalClaimedFees0 = totalClaimedFees0.add(fee0);
             totalClaimedFees1 = totalClaimedFees1.add(fee1);
@@ -296,8 +296,8 @@ contract MuteAmplifier is Ownable{

         // payout fee0 fee1
         if ((fee0 > 0 || fee1 > 0) && fee0 <= totalFees0 && fee
-            address(IMuteSwitchPairDynamic(lpToken).token0()).s
-            address(IMuteSwitchPairDynamic(lpToken).token1()).s
+            if (fee0 > 0) address(IMuteSwitchPairDynamic(lpToke
+            if (fee1 > 0) address(IMuteSwitchPairDynamic(lpToke

             totalClaimedFees0 = totalClaimedFees0.add(fee0);
             totalClaimedFees1 = totalClaimedFees1.add(fee1);
@@ -332,8 +332,8 @@ contract MuteAmplifier is Ownable{

             // payout fee0 fee1
```

```diff
                  if ((fee0 > 0 || fee1 > 0) && fee0 <= totalFees0 &&
-                     address(IMuteSwitchPairDynamic(lpToken).token0(
-                     address(IMuteSwitchPairDynamic(lpToken).token1(
+                     if (fee0 > 0) address(IMuteSwitchPairDynamic(lp
+                     if (fee1 > 0) address(IMuteSwitchPairDynamic(lp

                      totalClaimedFees0 = totalClaimedFees0.add(fee0)
                      totalClaimedFees1 = totalClaimedFees1.add(fee1)
```

[N-01] Remove `require` statements that are always true

The following two `require` statements always pass:

https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/dao/dMute.sol#L99-L100

`lock_info.amount` and `lock_info.tokens_minted` are of type `uint256` so they cannot be `< 0`.

In fact the check that `tokens_to_mint > 0` in the `LockTo` function even ensures that `lock_info.amount` and `lock_info.tokens_minted` are greater `0`.

[N-02] Remove `SafeMath` library

All 3 in-scope contracts use the `SafMath` library for simple addition, subtraction, multiplication and division.

This causes an unnecessary overhead since beginning from Solidity version `0.8.0` arithemtic operations revert by default on overflow / underflow.

By looking into the implementation of the `SafeMath` library you can also see that it is just a wrapper around the basic arithmatic operations and does not add any checks (at least for the functions used in the in-scope contracts) (Link).

[N-03] Event parameter names are messed up

In the `MuteBond` contract, the following events are defined:

Link

```
    event BondCreated(uint deposit, uint payout, address depositor,
    event BondPriceChanged(uint internalPrice, uint debtRatio);
    event MaxPriceChanged(uint _price);
    event MaxPayoutChanged(uint _price);
    event EpochDurationChanged(uint _payout);
    event BondLockTimeChanged(uint _duration);
    event StartPriceChanged(uint _lock);
```

Some of the parameter names do not accurately describe the variable that is actually emitted. E.g. for the `EpochDurationChanged` event, it is the new epoch duration that is emitted and not a `_payout` variable.

There is also an inconsistency with the `MaxPayoutChanged` and `StartPriceChanged` events. So I recommend to use more descriptive names in the event emissions for these 3 events.

[N-04] Event is never emitted

There are two events defined that are never emitted. They can be removed to make the code cleaner.

https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/bonds/MuteBond.sol#L13

https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/amplifier/MuteAmplifier.sol#L22

[N-05] Move `payoutFor` calculation into `else` block

https://github.com/code-423n4/2023-03-mute/blob/4d8b13add2907b17ac14627cfa04e0c3cc9a2bed/contracts/bonds/MuteBond.sol#L155

In case the `if` block is entered, `payout` is calculated twice.

Therefore the `payoutFor` call can be moved into the `else` block to clean up the code and save a little bit of Gas.

Fix:

```
diff --git a/contracts/bonds/MuteBond.sol b/contracts/bonds/Mute
index 96ee755..b462992 100644
--- a/contracts/bonds/MuteBond.sol
+++ b/contracts/bonds/MuteBond.sol
@@ -152,11 +152,12 @@ contract MuteBond {
      */
     function deposit(uint value, address _depositor, bool max_b
         // amount of mute tokens
-        uint payout = payoutFor( value );
+        uint payout;
         if(max_buy == true){
           value = maxPurchaseAmount();
           payout = maxDeposit();
         } else {
+          payout = payoutFor( value );
           // safety checks for custom purchase
           require( payout >= ((10**18) / 100), "Bond too small'
           require( payout <= maxPayout, "Bond too large"); // s
```

[N-06] Remove redundant check in `stake` function

The following check is redundant:

[Link]

```
require(IERC20(muteToken).balanceOf(address(this)) > 0, "MuteAmp
```

This check is redundant because before this check there is [another check] that `startTime!=0` which means the `initializeDeposit` function has been called which ensures the `MUTE` balance is not zero.

There are edge cases where the current check would apply, e.g. when staking occurs after the `endTime`. But the current check is not sufficient in this case because there could just be a little excess `MUTE` balance in the contract and the user would still not get rewards. So I recommend to remove the existing check and the edge cases will be addressed by the other changes I propose in this report.

Gas Optimizations

For this audit, 3 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **0xA5DF** received the top score from the judge.

The following wardens also submitted reports: evan and chaduke.

[G-01] Remove `SafeMath` library

Gas saved: 2.1K for each tx that uses `SafeMath` + 100 units per call.

It seems like safe math doesn't serve any purpose here and it can simply be replaced with normal math operands. The usage of library is pretty expensive since it makes a delegate call to an external address.

[G-02] Make `MuteAmplifier` variables immutable when possible

Gas saved: up to 20K per tx

The following variables can be immutable:

- `lpToken`
- `muteToken`
- `dToken`
- `_stakeDivisor`
- `management_fee`
- `treasury`

[G-03] Refactor `initializeDeposit()` into the constructor to make more variables immutable

Gas saved: ~7K per tx

The logic of `initializeDeposit()` cen be integrated into the constructor to make the following variables immutable:

- `totalRewards`
- `startTime`
- `endTime`

The only issue is sending tokens before the contract creation.

This can be solved in one of the following ways:

- Pre-calculate the address of the contract and approve the tokens to the address before creating it
- Add a deployer contract that will create the `MuteAmplifier` contract. The deployer contract will hold the tokens and will contain a callback that when called would send the tokens to the sender. The callback will revert if it'll be called at any time except during deployment (it'll use a storage variable to indicate if we're during a deployment or not)

[G-04] Use `lock_index` to look for empty elements

Gas saved: 2.2K * (amount of non-empty elements)

At dMute.redeemTo() the function iterates through the entire array to find empty elements, this means the entire array is being read in search for the empty elements.

Instead, we can find the empty elements using the `lock_index` and iterate through them only.

[G-05] At `MuteAmplifier.update()` skip updating if done in the same block

Gas saved: A few thousands

When `update()` is being called a second time in the same block both the `_totalWeight` and `_mostRecentValueCalcTime` don't really change. But the logic to update them still run and uses a few thousands of gas units.

Skipping that when `sinceLastCalc` is zero can save that gas.

Additionally, the remaining logic regarding fees might be skipped too (not sure about that).

Anyways, if the `fee0` or `fee1` are zero, some of the logic can be skipped too and save gas.

[G-06] `MuteBond.deposit()` makes an unnecessary call to `payoutFor()` during max buy

Gas saved: a few hundreds probably

In the following code the `payoutFor()` is unnecessary when `max_buy` is true and therefore should be inside the else block.

```
            uint payout = payoutFor( value );
            if(max_buy == true){
              value = maxPurchaseAmount();
              payout = maxDeposit();
            } else {
```

[G-07] Storage variable caching

Gas saved: 100 per cache

The following variables can be cached into memory instead of being read (or written) twice:

Here 2 storage variables are being read twice:
[amplifier/MuteAmplifier.sol#L370-L373]()

```
            // current rewards based on multiplier
            reward = lpTokenOut.mul(_totalWeight.sub(_userWeighted[a
            // max possible rewards
            remainder = lpTokenOut.mul(_totalWeight.sub(_userWeighte
```

`_userStakedBlock` is being read twice here: [amplifier/MuteAmplifier.sol#L480-L481]()

```
            uint256 staked_block =  _userStakedBlock[account] == blc
```

`epochStart` is being read and written a few times: [bonds/MuteBond.sol#L185-L197]()

```
            // adjust price by a ~5% premium of delta
            uint timeElapsed = block.timestamp - epochStart;
            epochStart = epochStart.add(timeElapsed.mul(5).div(100))
            // safety check
            if(epochStart >= block.timestamp)
              epochStart = block.timestamp;

            // exhausted this bond, issue new one
            if(terms[epoch].payoutTotal == maxPayout){
                terms.push(BondTerms(0,0,0));
                epochStart = block.timestamp;
                epoch++;
            }
```

[G-08] `_applyReward()` doing `sstore`s twice when called from `stake()`

Gas saved: ~400

When `_applyReward()` is being called from `stake()` [the following lines]() are writing to variables that are then written again at `_stake()`:

```
            _totalStakeLpToken = _totalStakeLpToken.sub(lpTokenOut);
            _userStakedLpToken[account] = 0;

            _userAccumulated[account] = 0;
```

Rewriting `_applyReward()` so that it won't do those changes (e.g. pass a parameter to indicate whether it's being called from `stake()` or not) and instead doing them at `_stake()` can save gas.

[G-09] Replace string errors with custom errors

Gas saved per instance: Probably a few thousands for deployment + a few hundreds when the function reverts

Custom errors are cheaper both for deployment and reverts, since strings take up lots of bytes code while custom reverts just a single byte.

[G-10] `MuteBond`'s `epoch()` and `currentEpoch()` are the same

Gas saved: a few units per tx

`epoch()` and `currentEpoch()` are basically the same, each public/external function adds a conditional check for each tx (since the contract compares the call data against the function has to find the right one). Removing this duplication can save a few gas units.

[G-11] Unnecessary multiplication and division by 1e18

Gas saved: ~20 units (200+ units when using `SafeMath`)

In [the following line](#) the multiplication and division by 1e18 is unnecessary.

Instead, just make sure all multiplications are done before division, that will prevent rounding in the same manner.

```
-        uint256 base_tokens = _amount.mul(_lock_time.mul(10**1
+        uint256 base_tokens = _amount.mul(_lock_time).div(max_l
```

Disclosures

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

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

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

Top