SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	MagicSea
Prepared by:	Sherlock
Lead Security Expert:	PUSH0
Dates Audited:	July 1 - July 11, 2024
Prepared on:	August 30, 2024

Introduction

MagicSea is the top Decentralized Exchange on IOTA & Shimmer EVM. Securely trade, stake & farm tokens to earn rewards on the most trusted DEX & NFT Marketplace. The audit is focusing on our MasterChef, Rewarder, our Magic LUM staking pool and Voting & Bribing System.

Scope

Repository: metropolis-exchange/magicsea-staking

Branch: main

Commit: fb97c8c92f6bca93d1a3184be625712c2fd3c994

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
9	7

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

santipu_
iamnmt
KupiaSec
dany.armstrong90
ChinmayF
PUSH0
jsmi
slowfi
scammed
0xAnmol
pashap9990
araj
kmXAdam
ydlee
HonorLt
BlockBusters
Honour
utsav
aslanbek
robertodf
Silvermist
rsam_eth
PeterSR
dhank
chaduke
blockchain555
0xAsen
0xWhitehat
excalibor
coffiasd
Reentrants
novaman33
qmdddd
minhquanym
sh0velware
web3pwn
dimulski
Aymen0909
0uts1der
zarkk01
DPS
LeFy
Yashar
oualidpro
radin200
blackhole
BengalCatBalu
neon2835
walter
Smacaud
karsar
sheep
Naresh
jah
Ryonen
AuditorPraise
bbl4de
jennifer37
tedox
anonymousjoe
Hunter
WildSniper
snapishere
gkrastenov
cocacola
0xpranav
t.aksoy
PNS
0xboriskataa
nikhil840096
sajjad
0xR360
DMoore
MrCrowNFT
Yanev
0xc0ffEE
TessKimy
neogranicen
.-..—.....-.
fibonacci
dev0cloo
eLSeR17
Gowtham_Ponnana
luke
John_Femi
Bauchibred
Tri-pathi
ZanyBonzy
pinalikefruit
StraawHaat
0xNilesh
MSaptarshi
rbserver
dod4ufn
gajiknownnothing
Vancelot
yixxas
NoOne
4rdiii
Ericselvig
typicalHuman
hunter_w3b
PASCAL
KiroBrejka

Issue H-1: Non-functional vote() if there is one bribe rewarder for this pool

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/39

Found by

0uts1der, 0xAsen, 0xboriskataa, 0xc0ffEE, AuditorPraise, Aymen0909, BengalCatBalu, BlockBusters, ChinmayF, DPS, HonorLt, Honour, Hunter, KupiaSec, PNS, PeterSR, Reentrants, Ryonen, Silvermist, WildSniper, Yanev, Yashar, araj, blackhole, blockchain555, coffiasd, dany.armstrong90, dhank, dimulski, gkrastenov, iamnmt, jennifer37, jsmi, kmXAdam, nikhil840096, pashap9990, radin200, robertodf, rsam_eth, scammed, slowfi, snapishere, t.aksoy, utsav, web3pwn

Summary

Permission check in BribeRewarder::deposit(), this will lead to vote() function cannot work if voted pool has any bribe rewarder.

Vulnerability Detail

When people vote for one pool, there may be some extra rewards provided by bribe rewarders. When users vote for one pool with some bribe rewarders, voter contract will call bribe rewarder's `deposit` function. However, in bribe rewarder's `deposit()` function, there is one security check, the caller should be the NFT's owner, which is wrong. Because the voter contract call bribe rewarder's `deposit()`, msg.sender is voter contract, not the owner of NFT. This will block all vote() transactions if this votes pool has any bribe rewarder.

```
    function vote(uint256 tokenId, address[] calldata pools, uint256[] calldata
↪   deltaAmounts) external {
        ......
        IVoterPoolValidator validator = _poolValidator;
        for (uint256 i = 0; i < pools.length; ++i) {
          ......
            uint256 deltaAmount = deltaAmounts[i];
            // per user account
            _userVotes[tokenId][pool] += deltaAmount;
            // per pool account
            _poolVotesPerPeriod[currentPeriodId][pool] += deltaAmount;
            // @audit_fp should we clean the _votes in one new vote period ,no
↪   extra side effect found
            if (_votes.contains(pool)) {
                _votes.set(pool, _votes.get(pool) + deltaAmount);
```


SHERLOCK

```
            } else {
                _votes.set(pool, deltaAmount);
            }
            // bribe reward will record voter
@==>    _notifyBribes(_currentVotingPeriodId, pool, tokenId, deltaAmount); //
↪   msg.sender, deltaAmount);
        }
        ......
    }
    function _notifyBribes(uint256 periodId, address pool, uint256 tokenId,
↪   uint256 deltaAmount) private {
        IBribeRewarder[] storage rewarders = _bribesPerPriod[periodId][pool];
        for (uint256 i = 0; i < rewarders.length; ++i) {
            if (address(rewarders[i]) != address(0)) {
                // bribe rewarder will record vote.
                rewarders[i].deposit(periodId, tokenId, deltaAmount);
                _userBribesPerPeriod[periodId][tokenId].push(rewarders[i]);
            }
        }
    }
    function deposit(uint256 periodId, uint256 tokenId, uint256 deltaAmount)
↪   public onlyVoter {
        _modify(periodId, tokenId, deltaAmount.toInt256(), false);

        emit Deposited(periodId, tokenId, _pool(), deltaAmount);
    }
    function _modify(uint256 periodId, uint256 tokenId, int256 deltaAmount, bool
↪   isPayOutReward)
        private
        returns (uint256 rewardAmount)
    {
        // If you're not the NFT owner, you cannot claim
        if (!IVoter(_caller).ownerOf(tokenId, msg.sender)) {
            revert BribeRewarder__NotOwner();
        }
```

Poc

When alice tries to vote for one pool with one bribe rewarder, the transaction will be reverted with the reason 'BribeRewarder__NotOwner'

```
function testPocVotedRevert() public {
    vm.startPrank(DEV);
    ERC20Mock(address(_rewardToken)).mint(address(ALICE), 100e18);
    vm.stopPrank();
    vm.startPrank(ALICE);
```

```
    rewarder1 =
↪   BribeRewarder(payable(address(factory.createBribeRewarder(_rewardToken,
↪   pool))));
    ERC20Mock(address(_rewardToken)).approve(address(rewarder1), 20e18);
    // Register
    //_voter.onRegister();
    rewarder1.fundAndBribe(1, 2, 10e18);
    vm.stopPrank();
    // join and register with voter
    // Create position at first
    vm.startPrank(ALICE);
    // stake in mlum to get one NFT
    _createPosition(ALICE);

    vm.prank(DEV);
    _voter.startNewVotingPeriod();
    vm.startPrank(ALICE);
    _voter.vote(1, _getDummyPools(), _getDeltaAmounts());
    // withdraw this NFT
    vm.stopPrank();
}
```

Impact

vote() will be blocked for pools which owns any bribe rewarders.

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/rewarders/BribeRewarder.sol#L143-L147
https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/rewarders/BribeRewarder.sol#L260-L269

Tool used

Manual Review

Recommendation

This security check should be valid in claim() function. We should remove this check from deposit().

Discussion

0xSmartContract

The internal `_modify()` call of the `deposit()` function checks the identity of the token owner; this fails because `msg.sender` is the voter contract, not the owner.So it is impossible to vote for pools that have bribe rewarders.

0xHans1

PR: https://github.com/metropolis-exchange/magicsea-staking/pull/13

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/13

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-2: A voter lose bribe rewards if another voter voted before claim.

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/52

Found by

0xAsen, 0xWhitehat, ChinmayF, Honour, KupiaSec, PeterSR, Silvermist, araj, blockchain555, dhank, excalibor, jsmi, kmXAdam, pashap9990, rsam_eth, slowfi, utsav

Summary

A voter lose bribe rewards if another voter voted before claim.

Vulnerability Detail

This problem is related to design architecture. In `BribeRewarder.sol`, the `_lastUpdateTimestamp` is used to calculate the unclaimed rewards for `periodId`, but it is not dependent on `periodId`. Therefore, once `_lastUpdateTimestamp` has been updated to the next period, there is no way to calculate the unclaimed rewards for the previous period.

The following is the modified test code for PoC.

```
function testDepositMultiple() public {
    ERC20Mock(address(rewardToken)).mint(address(this), 20e18);
    ERC20Mock(address(rewardToken)).approve(address(rewarder), 20e18);

    rewarder.fundAndBribe(1, 2, 10e18);

    _voterMock.setCurrentPeriod(1);
    _voterMock.setStartAndEndTime(0, 100);

    // time: 0
    vm.warp(0);
    vm.prank(address(_voterMock));
    rewarder.deposit(1, 1, 0.2e18);

    assertEq(0, rewarder.getPendingReward(1));

    // time: 50, seconds join
    vm.warp(50);
    vm.prank(address(_voterMock));
    rewarder.deposit(1, 2, 0.2e18);
```

```
    // time: 100
    vm.warp(100);
    _voterMock.setCurrentPeriod(2);
    _voterMock.setStartAndEndTime(0, 100);
    _voterMock.setLatestFinishedPeriod(1);

    // @audit-info next period started
    vm.warp(150);

    // 1 -> [0,50] -> 1: 0.5
    // 2 -> [50,100] -> 1: 0.25 + 0.5, 2: 0.25

    assertEq(7500000000000000000, rewarder.getPendingReward(1));
    assertEq(2500000000000000000, rewarder.getPendingReward(2));

    // @audit-info Another voter votes before claim.
    vm.prank(address(_voterMock));
    rewarder.deposit(2, 3, 0.1e18);

    // @audit-info The expected rewards decreased much
    assertEq(5000000000000000000, rewarder.getPendingReward(1));
    assertEq(0, rewarder.getPendingReward(2));

    vm.prank(alice);
    rewarder.claim(1);

    vm.prank(bob);
    rewarder.claim(2);

    // @audit-info The claimed rewards decreased too.
    assertEq(5000000000000000000, rewardToken.balanceOf(alice));
    assertEq(0, rewardToken.balanceOf(bob));

    assertEq(7500000000000000000, rewardToken.balanceOf(alice), "balance of
↪   alice should be 75e17 but 50e17");
    assertEq(2500000000000000000, rewardToken.balanceOf(bob), "balance of bob
↪   should be 25e17 but 0");
}
```

The claimed rewards amount of alice and bob for 1st period are originally `75e17` and `25e17`, respectively. But if a voter votes for 2nd period before alice and bob claim their rewards for 1st period, the claimed rewards amount of alice and bob will be decreased to `50e17` and `zero`, respectively. It means that the rewards for [50,100] of 1st period are will not be claimed.

And the following is the test command and result.

```
$ forge test --match-test testDepositMultiple

Failing tests:
Encountered 1 failing test in test/BribeRewarder.t.sol:BribeRewarderTest
[FAIL. Reason: balance of alice should be 75e17 but 50e17: 7500000000000000000
 ↪  != 5000000000000000000] testDepositMultiple() (gas: 767761)
```

As shown above, if anyone votes before alice and bob claim their rewards, the rewards of alice and bob will be decreased.

Impact

Voters lose bribe rewards if another voter voted before claim. And such cases can occur frequently enough.

Code Snippet

- magicsea-staking/src/rewarders/BribeRewarder.sol#L65
- magicsea-staking/src/rewarders/BribeRewarder.sol#L260-L298

Tool used

Manual Review

Recommendation

Change the `_lastUpdateTimestamp` state variable to be dependent on `periodId`. For instance, change it to mapping variable such as `mapping(uint256 periodId => uint256) _lastUpdateTimestamp;`.

Discussion

0xSmartContract

High severity because this will occur without an attacker and between all Bribe Rewards, resulting in reward loss for all voters. A side effect of this is that rewards sent to Bribe Rewards will be stuck in the contract forever.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/20

chinmay-farkya

Escalate

The first thing to remember is that this issue is different from undistributed/ unaccrued rewards getting stuck in the contract due to many reasons like : no voting, precision loss etc. That is a different set of issue reported elsewhere and is meant for “undistributed rewards”. Whereas the current issue set #52 describes a bug that the rewards that users had “already accrued” can not be claimed by them, so this is a problem of claiming accrued rewards rather than undistributed rewards. Fixing either of them will not fix the other, so these are two different sets of issues.

Now lets look at some issues wrongly duped with this #52

#303 is not a duplicate and is invalid. When the first call is made totalRewards returned = 0 because totalSupply = 0 and in this first call while setting the reward parameters rewarder.lastUpdateTimestamp gets updated to the startTimestamp of the reward cycle. So the assertion made in this submission “rewards calculated for time when lastUpdateTimestamp was zero will be huge” is wrong. For all later calls, this actual non-zero lastUpdateTimestamp will be used, so the calculation fo rewards is correct.

#397 is not a duplicate and is invalid because even though the totalRewards calculation is wrong it has no real impact as the “totalRewards” are manifested into storage using rewarder.update which returns 0 because the oldTotalSupply was zero in the mentioned case of first voter in a briberewarder’s bribe periods. Have a look at rewarder.update here. So oldTotalSupply and oldBalance being zero, this rewards calculation is completely ignored i current logic and no one can earn extra rewards as claimed in the issue. Hence invalid.

#496 is a duplicate of #164 (note :also read the escalation on #164 as this issue #496 overall describes “undistributed rewards will be stuck” is a duplicate of set 2 of the issues mentioned in that escalation) because some of the issues in set 2 of #164 also describe precision loss as a way of having funds stuck in the contract. For example see #172 also mentioned in the escalation on #164.

#582 is also not a duplicate and is invalid because the bug has no impact due to similar reason as #397 as explained above.

#588 is invalid due to the same reasons as #397. Ping me if you want to discuss this bug more.

#607 is again invalid. It goes over the same assertion that rewards calculation will yield a huge value because lastUpdateTimestamp is a time in past before the period started, but misses the point that this bug never materializes due to how the rewarder2 library works. If the totalsupply before the call was zero (which means this is the first call) these “totalRewards” never accrue in the accDebtPerShare and hence are not distributed. On the next call, these are accrued correctly based on the time interval from this first call to the second call. So the accrued rewards

never includes the rewards calculated for time before the first bribe period started. Hence, similar to #397 and invalid.

#356 is not a duplicate of this but a duplicate of #164 set 2. It talks about a portion of rewards that was undistributed getting stuck in the BribeRewarder contract. See comment for #496 above.

sherlock-admin3

> Escalate
>
> The first thing to remember is that this issue is different from undistributed/ unaccrued rewards getting stuck in the contract due to many reasons like : no voting, precision loss etc. That is a different set of issue reported elsewhere and is meant for “undistributed rewards”. Whereas the current issue set #52 describes a bug that the rewards that users had “already accrued” can not be claimed by them, so this is a problem of claiming accrued rewards rather than undistributed rewards. Fixing either of them will not fix the other, so these are two different sets of issues.
>
> Now lets look at some issues wrongly duped with this #52
>
> #303 is not a duplicate and is invalid. When the first call is made totalRewards returned = 0 because totalSupply = 0 and in this first call while setting the reward parameters rewarder.lastUpdateTimestamp gets updated to the startTimestamp of the reward cycle. So the assertion made in this submission “rewards calculated for time when lastUpdateTimestamp was zero will be huge” is wrong. For all later calls, this actual non-zero lastUpdateTimestamp will be used, so the calculation fo rewards is correct.
>
> #397 is not a duplicate and is invalid because even though the totalRewards calculation is wrong it has no real impact as the “totalRewards” are manifested into storage using rewarder.update which returns 0 because the oldTotalSupply was zero in the mentioned case of first voter in a briberewarder’s bribe periods. Have a look at rewarder.update here. So oldTotalSupply and oldBalance being zero, this rewards calculation is completely ignored i current logic and no one can earn extra rewards as claimed in the issue. Hence invalid.
>
> #496 is a duplicate of #164 (note :also read the escalation on #164 as this issue #496 overall describes “undistributed rewards will be stuck” is a duplicate of set 2 of the issues mentioned in that escalation) because some of the issues in set 2 of #164 also describe precision loss as a way of having funds stuck in the contract. For example see #172 also mentioned in the escalation on #164.

> #582 is also not a duplicate and is invalid because the bug has no impact due to similar reason as #397 as explained above.
>
> #588 is invalid due to the same reasons as #397. Ping me if you want to discuss this bug more.
>
> #607 is again invalid. It goes over the same assertion that rewards calculation will yield a huge value because lastUpdateTimestamp is a time in past before the period started, but misses the point that this bug never materializes due to how the rewarder2 library works. If the totalsupply before the call was zero (which means this is the first call) these “totalRewards” never accrue in the accDebtPerShare and hence are not distributed. On the next call, these are accrued correctly based on the time interval from this first call to the second call. So the accrued rewards never includes the rewards calculated for time before the first bribe period started. Hence, similar to #397 and invalid.
>
> #356 is not a duplicate of this but a duplicate of #164 set 2. It talks about a portion of rewards that was undistributed getting stuck in the BribeRewarder contract. See comment for #496 above.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xSmartContract

#303 , #397 , #587 ; The are not a duplicate and is invalid (I aggre)

#582 ; This is not a duplicate and is invalid (I aggre) Issue #582 was related to an incorrect update inside the function _lastUpdateTimestamp _modify(). This was not actually a problem in the code as implemented. Issue #52 highlights a different and real issue in the design of the reward calculation system. This issue relates to how the single `_lastUpdateTimestamp` variable is used across different periods, which could lead to a loss of rewards for some users.

2 different sets ; I don't agree

chinmay-farkya

@0xSmartContract

> 2 different sets ; I don't agree

I think there is some misunderstanding. I did not say that this issue has 2 sets. I only pointed out specific issues which are invalid and wrongly duped with this. Rest assured, this is only a single issue and not two sets, but please go over the invalid ones and remove them from the dups (list mentioned in original escalation).

0xSmartContract

> @0xSmartContract
>
> > 2 different sets ; I don't agree
> >
> > I think there is some misunderstanding. I did not say that this issue has 2 sets. I only pointed out specific issues which are invalid and wrongly duped with this. Rest assured, this is only a single issue and not two sets, but please go over the invalid ones and remove them from the dups (list mentioned in original escalation).

I already said "I don't agree" for the set part, so my "I aggre" here can be taken into account.

https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/52#issuecomment-2267165669

WangSecurity

I agree with the escalation and plan to accept it. Here are the changes I'm going to apply:

1. Invalidate #303, #397, #582, #588, #607.
2. #496 is escalated as well, I agree it's not a duplicate of #52 and will be considered during the #164 escalation.
3. Same for #356.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- chinmay-farkya: accepted

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-3: Wrong call order for `setTopPoolIdsWithWeights`, resulting in wrong distribution of rewards

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/107

Found by

PUSH0, dany.armstrong90, iamnmt, jsmi

Summary

Per the Sherlock rules:

> If the protocol team provides specific information in the README or CODE COMMENTS, that information stands above all judging rules.

The Masterchef contract allows people to stake an admin-selected token in farms to earn LUM rewards. Each two weeks, MLUM stakers can vote on their favorite pools, and the top pools will earn LUM emissions according to the votes. Admin has to call `setTopPoolIdsWithWeights` to set those votes and weights to set the reward emission for the next two weeks.

Per the documented call order for `setTopPoolIdsWithWeights`:

```
/**
* @dev Set farm pools with their weight;
*
* WARNING:
* Caller is responsible to updateAll oldPids on masterChef before using this
 ↪  function
* and also call updateAll for the new pids after.
*
* @param pids - list of pids
* @param weights - list of weights
*/
```

We show that this call order is wrong, and will result in wrong rewards distribution.

Vulnerability Detail

There is a global parameter `lumPerSecond`, set by the admin. Whenever `updateAll` is called for a set of pools:

- Let the total weight of all votes across all top pools be `totalWeight`
- Let the **current** weight of a pool Pid be `weightPid`. This weight can be set by the admin using `setTopPoolIdsWithWeights`

- Pool Pid will earn `totalLumRewardForPid = (lumPerSecond * weightPid / totalWeight) * (elapsed_time)`, i.e. each second it earns `lumPerSecond` times its percentage of voted weight `weightPid` across the total weight all top pools `totalWeight`.

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/MasterchefV2.sol#L522-L525

Now, the function `updateAll` does the following:

- For each pool, fetch its weight and calculate its `totalLumRewardForPid` since last update
- Mint that calculated amount of LUM
- `updateAccDebtPerShare` i.e. distribute rewards since the last updated time

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/MasterchefV2.sol#L526-L528

Per the code comments, the admin is responsible for calling `updateAll()` on the old pools before calling `setTopPoolIdsWithWeights()` for the new pools, and *then* calling `updateAll()` on the new pools.

We claim that, using this call order, a pool will be wrongly updated if it's within the set `newPid` but not in `oldPid`, and the functions are called with this order. Take this example.

PoC

Let LUM per second = 1. We assume all farms were created and registered at time 0:

- At time 1000: there are two pools, A and B making it into the top pools. Weight = 1 both.
  - `updateAll` for oldPid. There are no old Pids
  - `setTopPoolIdsWithWeights`: Pool A and pool B now have weight = 1.
  - `updateAll` for newPid (A and B). 1000 seconds passed, each pool accrued 500 LUM for having 50% weight despite just making it into the top weighted pools
- At time 2000: a new pool C makes it into the pool, while pool B is no longer in the top. Weight = 1 both.
  - `updateAll` for oldPid (A and B).
    * For pool A, 1000 seconds passed. It earns 500 LUM
    * For pool B, 1000 seconds passed. It earns 500 LUM

    - `setTopPoolIdsWithWeights`: Pool A and pool C now have weight = 1.
    - `updateAll` for newPid (A and C).
        * For pool A, 0 seconds passed. It earns 0 LUM
        * For pool C, 2000 seconds passed. It earns 1000 LUM

The end result is that, at time 2000:

- Pool A accrued 1000 LUM
- Pool B accrued 1000 LUM
- Pool C accrued 1000 LUM

Where the correct result should be:

- Pool A accrued 500 LUM, it was in the top pools in time 1000 to 2000
- Pool B accrued 500 LUM, it was in the top pools in time 1000 to 2000
- Pool C accrued 0 LUM, it only started being in the top pools from timestamp 2000

In total, 3000 LUM has been distributed from timestamps 1000 to 2000, despite the emission rate should be 1 LUM per second. In fact, LUM has been wrongly distributed since timestamp 1000, as both pool A and B never made it into the top pools but still immediately accrued 500 LUM each.

This is because if a pool is included in an `updateAll` call **after** its weight has been set, its last updated timestamp is still in the past. Therefore when `updateAll` is called, the new weights are applied across the entire interval since it was last updated (i.e. a far point in the past).

Coded PoC

We provide a coded PoC to prove the impact of timestamp 1000. We add two farms A and B. LUM per second is set to 1000 wei per second. We also have a single staker Alice depositing into farm A.

We have two tests to compare the results:

- Both tests set the pool weights at time 1000, then output Alice's pending reward right after that.
- There are no `oldPids`, so there's no need to call `updateAll()` on anything before setting weights.
- In one test, we call `updateAll(newPids)` *after* calling `setTopPoolIdsWithWeights()`.

- In the other test, we call `updateAll(newPids)` *before* calling `setTopPoolIdsWithWeights()`.

We output the pending rewards of Alice for comparison.

First, change the function `mint()` of contract `MockERC20` to be the following:

```
function mint(address _to, uint256 _amount) external returns (uint256) {
    _mint(_to, _amount);
    return _amount;
}
```

Then, create a new test file `MasterChefTest.t.sol`:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

import "forge-std/Test.sol";

import "openzeppelin-contracts-upgradeable/access/OwnableUpgradeable.sol";
import {SafeERC20, IERC20} from "openzeppelin/token/ERC20/utils/SafeERC20.sol";

import "../src/transparent/TransparentUpgradeableProxy2Step.sol";

import "openzeppelin/token/ERC721/ERC721.sol";
import "openzeppelin/token/ERC20/ERC20.sol";
import {ERC20Mock} from "./mocks/ERC20.sol";
import {MasterChef} from "../src/MasterChefV2.sol";
import {MlumStaking} from "../src/MlumStaking.sol";
import "../src/Voter.sol";
import "../src/rewarders/BaseRewarder.sol";
import "../src/rewarders/MasterChefRewarder.sol";
import "../src/rewarders/RewarderFactory.sol";
import {IVoter} from "../src/interfaces/IVoter.sol";
import {ILum} from "../src/interfaces/ILum.sol";
import {IRewarderFactory} from "../src/interfaces/IRewarderFactory.sol";

contract MasterChefV2Test is Test {
    address payable immutable DEV = payable(makeAddr("dev"));
    address payable immutable ALICE = payable(makeAddr("alice"));
    address payable immutable BOB = payable(makeAddr("bob"));

    Voter private _voter;
    MlumStaking private _pool;
    MasterChef private _masterChefV2;
```


SHERLOCK

```
    ERC20Mock private farmA;
    ERC20Mock private farmB;
    ERC20Mock private farmC;

    MasterChefRewarder rewarderPoolA;
    MasterChefRewarder rewarderPoolB;
    MasterChefRewarder rewarderPoolC;

    RewarderFactory factory;

    ERC20Mock private _stakingToken;
    ERC20Mock private _rewardToken;
    ERC20Mock private _lumToken;

    uint256[] pIds;
    uint256[] weights;

    function setUp() public {
        vm.prank(DEV);
        _stakingToken = new ERC20Mock("MagicLum", "MLUM", 18);

        vm.prank(DEV);
        _rewardToken = new ERC20Mock("USDT", "USDT", 6);

        vm.prank(DEV);
        address poolImpl = address(new MlumStaking(_stakingToken, _rewardToken));

        _pool = MlumStaking(
            address(
                new TransparentUpgradeableProxy2Step(
                    poolImpl, ProxyAdmin2Step(address(1)),
↪   abi.encodeWithSelector(MlumStaking.initialize.selector, DEV)
                )
            )
        );

        address factoryImpl = address(new RewarderFactory());
        factory = RewarderFactory(
            address(
                new TransparentUpgradeableProxy2Step(
                    factoryImpl,
                    ProxyAdmin2Step(address(1)),
                    abi.encodeWithSelector(
                        RewarderFactory.initialize.selector, address(this), new
↪   uint8[](0), new address[](0)
                    )
                )
```


SHERLOCK

```
            )
        );

        vm.prank(DEV);
        _lumToken = new ERC20Mock("Lum", "LUM", 18);
        farmA = new ERC20Mock("Farm A", "FARM A", 18);
        farmB = new ERC20Mock("Farm B", "FARM B", 18);
        farmC = new ERC20Mock("Farm C", "FARM C", 18);

        vm.prank(DEV);
        address masterChefImp = address(new MasterChef(ILum(address(_lumToken)),
↪   _voter, factory,DEV,1));

        _masterChefV2 = MasterChef(
            address(
                new TransparentUpgradeableProxy2Step(
                    masterChefImp, ProxyAdmin2Step(address(1)),
↪   abi.encodeWithSelector(MasterChef.initialize.selector, DEV,DEV)
                )
            )
        );
        vm.prank(DEV);
        _masterChefV2.setLumPerSecond(1000);
        vm.prank(DEV);
        _masterChefV2.setMintLum(true);

        //add 3 farms
        vm.prank(DEV);
        _masterChefV2.add(farmA,IMasterChefRewarder(address(0)));
        vm.prank(DEV);
        _masterChefV2.add(farmB,IMasterChefRewarder(address(0)));
        vm.prank(DEV);
        _masterChefV2.add(farmC,IMasterChefRewarder(address(0)));

        vm.prank(DEV);
        address voterImpl = address(new Voter(_masterChefV2, _pool, factory));

        _voter = Voter(
            address(
                new TransparentUpgradeableProxy2Step(
                    voterImpl, ProxyAdmin2Step(address(1)),
↪   abi.encodeWithSelector(Voter.initialize.selector, DEV)
                )
            )
        );

        vm.prank(DEV);
```


```
        _voter.updateMinimumLockTime(2 weeks);

        vm.prank(DEV);
        factory.setRewarderImplementation(
            IRewarderFactory.RewarderType.MasterChefRewarder,
↪   IRewarder(address(new MasterChefRewarder(address(_masterChefV2))))
        );

        vm.prank(DEV);
        _masterChefV2.setVoter(_voter);
    }

    //SetPoolWeightsTest Correct
    function testSetPoolWeightsCorrect() public {
        pIds.push(0);
        pIds.push(1);
        weights.push(500);
        weights.push(500);

        farmA.mint(ALICE, 2 ether);

        vm.prank(ALICE);
        farmA.approve(address(_masterChefV2), 1 ether);
        vm.prank(ALICE);
        _masterChefV2.deposit(0, 1 ether);

        skip(1000);
        vm.prank(DEV);
        _masterChefV2.updateAll(pIds);

        vm.prank(DEV);
        _voter.setTopPoolIdsWithWeights(pIds,weights);

        (uint256[] memory lumRewards,IERC20[] memory tokens,uint256[] memory
↪   extraRewards) = _masterChefV2.getPendingRewards(ALICE, pIds);
        console.log("Alice rewards correct:");
        console.log(lumRewards[0]);
    }

    //SetPoolWeightsTest Wrong
    function testSetPoolWeightsWrong() public {
        pIds.push(0);
        pIds.push(1);
        weights.push(500);
        weights.push(500);

        farmA.mint(ALICE, 2 ether);
```


```
        vm.prank(ALICE);
        farmA.approve(address(_masterChefV2), 1 ether);
        vm.prank(ALICE);
        _masterChefV2.deposit(0, 1 ether);

        skip(1000);
        vm.prank(DEV);
        _voter.setTopPoolIdsWithWeights(pIds,weights);

        vm.prank(DEV);
        _masterChefV2.updateAll(pIds);

        (uint256[] memory lumRewards,IERC20[] memory tokens,uint256[] memory
↪   extraRewards) = _masterChefV2.getPendingRewards(ALICE, pIds);
        console.log("Alice rewards wrong:");
        console.log(lumRewards[0]);
    }
}
```

And the results are:

As shown, in the "correct" test, Alice has not accrued any rewards right after the new weights are set. However, in the "wrong" test, Alice accrues 499999 reward units right after setting.

Impact

Pools that have just made it into the top pools will have already accrued rewards for time intervals it wasn't in the top pools. Rewards are thus severely inflated.

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/Voter.sol#L250-L260

Tool used

Manual Review

Recommendation

All of the pools (oldPids and newPids) should be updated, only then weights should be applied.

In other words, the correct call order should be:

- `updateAll` should be called for **all** pools within oldPid or newPid.
- `setTopPoolIdsWithWeights` should then be called.

Additionally, we think it might be better that `setTopPoolIdsWithWeights` itself should just call `updateAll` for all (old and new) pools before updating the pool weights, or at least validate that their last updated timestamp is sufficiently fresh.

Discussion

0xSmartContract

Failure to follow the correct order of calls of the `setTopPoolIdsWithWeights` function in the `Masterchefv2.sol` contract causes pools to accumulate incorrect LUM rewards. The description of the function states that the `updateAll` function should be called for old repositories and then called again for new repositories.

However, if this call order is not followed, new pools will also accumulate rewards for periods before entering the top pools. This leads to over distribution of rewards and a serious inflation effect.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/22

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-4: Voters will lose all bribe rewards forever if they do not claim their rewards after the last bribing period

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/164

Found by

ChinmayF, KupiaSec, iamnmt, scammed, slowfi

Summary

The `claim()` function in `BribeRewarder` is used to claim the rewards associated with a tokenID across all bribe periods that have ended : it iterates over all the voting periods starting from the bribe rewarder's `_startVotingPeriod` upto the last period that ended according to the Voter.sol contract, and collects and sends rewards to the NFT's owner.

The issue is that if the voter(ie. tokenID owner who earned bribe rewards for one or more bribe periods) does not claim his rewards by the lastVotingPeriod + 2, then all his unclaimed rewards for all periods will be lost forever.

Vulnerability Detail

Lets walk through an example to better understand the issue. Even though the issue occurs in all other cases, we are assuming that the voting period has just started to make it easy to understand.

1. The first voting period is about to start in the next block. The bribe provider deploys a bribe rewarder and registers it for a pool X for voting periods 1 to 5. ie. the startVotingPeriod in the BribeRewarder.sol = 1 and lastVotingPeriod = 5.
2. The first voting period starts in voter.sol. Users start voting for pool X and the BribeRewarder keeps getting notified and storing the rewards data in respective rewarder data structure (see here and here)
3. All 5 voting periods have ended. User voted in all voting periods and got a reward stored for all 5 bribe periods in the BribeRewarder contract. Now when he claims via claim(), he can get all his rewards.
4. Assume that the 6th voting period has ended. Still if the user calls claim(), he will get back all his rewards. His 6th period rewards will be empty but it does not revert.
5. Assume that the 7th voting period has ended. Now if the user calls claim(), his call will revert and from now on, he will never be able to claim any of his unclaimed rewards for all periods.

The reason of this issue is this :

```
function claim(uint256 tokenId) external override {
    uint256 endPeriod = IVoter(_caller).getLatestFinishedPeriod();
    uint256 totalAmount;

    for (uint256 i = _startVotingPeriod; i <= endPeriod; ++i) {
        totalAmount += _modify(i, tokenId, 0, true);
    }

    emit Claimed(tokenId, _pool(), totalAmount);
}
```

The claim function is the only way to claim a user's rewards after he has voted. This iterates over the voting periods starting from the `_startVotingPeriod` (which is equal to 1 in our example).

This loop's last iteration is the latest voting period that might have ended on the voter contract (regardless of if it was a declared as a bribe period in our own BribeRewarder since voting periods will be a forever going thing and we only want to reward upto a limited set of periods, defined by `BribeRewarder:_lastVotingPeriod`).

Lets see the `Voter.getLatestFinishedPeriod()` function :

```
function getLatestFinishedPeriod() external view override returns (uint256) {
    if (_votingEnded()) {
        return _currentVotingPeriodId;
    }
    if (_currentVotingPeriodId == 0) revert IVoter__NoFinishedPeriod();
    return _currentVotingPeriodId - 1;
}
```

Now if suppose the 6th voting period is running, it will return 5 as finished. if 7th is running, it will return 6, and if 8th period has started, it will return 7.

Now back to `claim => _modify`. It fetches the rewards data for that period in the `_rewards` array, which has (lastID - startID) + 2 elements. (see here). In our case, this array will consist of 6 elements (startId = 1 and lastID = 5).

Now when we see how it is fetching the reward data using periodID, it is using the value returned by `_indexByPeriodId()` as the index of the array.

```
function _indexByPeriodId(uint256 periodId) internal view returns (uint256) {
    return periodId - _startVotingPeriod;
}
```

So for a periodID = 7, this will return index 6.

Now back to the example above. When the 7th voting period has ended, getLatestFinishedPeriod() will return 7 and the claim function will try to iterate over all the periods that have ended. When the iteration comes to this last period = 7, the `_modify` function will try to read the array element at index 6 (again we can see this clearly here)

But now it will revert with index out of bounds, because the array only has 6 elements from index 0 to index 5, so trying to access index element 6 in `_rewards` array will now always revert.

This means that after 2 periods have passed after the last bribing period, no user can ever claim any of their rewards even if they voted for all the periods.

Impact

No user will be able to claim any of their unclaimed rewards for any periods from the BribeRewarder, after this time. The rewards will be lost forever, and a side effect of this is that these rewards will remain stuck in the BribeRewarder. But the main impact is the complete loss of rewards of many users.

High severity because users should always get their deserved rewards, and many users could lose rewards this way at the same time. The damage to the individual user depends on how many periods they didn't claim for and how much amount they used for voting, which could be a very large amount.

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/rewarders/BribeRewarder.sol#L159

Tool used

Manual Review

Recommendation

The solution is simple : in the claim function, limit the `endPeriod` used in the loop by the `_lastVotingPeriod` of a particular BribeRewarder.

```
uint256 endPeriod = IVoter(_caller).getLatestFinishedPeriod();
if (endPeriod > _lastVotingPeriod) endPeriod = _lastVotingPeriod;
```

Discussion

0xSmartContract

If the function to claim rewards in the `BribeRewarder` contract is not used within two periods after the last voting period, users will lose all rewards.

This may cause many users to lose their rewards, which will remain locked in the contract forever. Solution ; is to limit the request cycle to `_lastVotingPeriod`.

0xHans1

PR: https://github.com/metropolis-exchange/magicsea-staking/pull/14

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/12

Reentrants

Incorrect duplication, there are a few issues mentioned that should be separated:

1. Incorrect `endPeriod` assignment, which this issue elaborates on
2. Lost bribe rewards whilst there are 0 votes
3. Rewards can't be swept

They should be separated into 3 different issues, perhaps (2) and (3) can be combined as (3) indirectly solves (2).

slowfinance

I am afraid Reentrants is right.

chinmay-farkya

Escalate

This set of issues is actually two completely different sets of issues clubbed together incorrectly. One set is the “Voters will lose all bribe rewards forever if they do not claim their rewards after the last bribing period” ie. this issue #164 which emphasizes the loss of legit rewards `already earned by the voters`. In this case, the root cause and fix are different ie. voters will not be able to claim their rewards after certain time. In this case, the voters are affected.

Here are duplicates of #164 : Set 1 : #145, #501, #495, Note that three issues #591, #161, #371 also identify the same issue but the description of these issues is technically incorrect because they say that the array out-of-bound revert will start happening after the lastVotingPeriod +1 which is not true because the array actually gets a length of lastID - startId + 2, as an extra empty element is added to the rewards array due to the wrong(but non-impactful) logic here. This means that

while these three issues say revert will happen after lastId +1 period, the revert will actually only start happening after lastId + 2 periods as per the current logic. Thus, these three issues should be deemed invalid due to the incorrect description of the issue, as they could not identify the correct conditions in which the bug manifests.

Thus, set 1 has only four valid issues : #164, #145, #501, #495

While the other set emphasises on the rewards that were not earned by any voter (maybe due to zero votes etc.) and thus they remain stuck in the briber contract. In this set, the root cause is the non-existence of a sweep function to restore rewards that were ‘’’not earned by anyone’. In this case, the briber itself is affected.

Solving one of them doesn’t solve the other as the fixes and root cause are completely unrelated.

Set 2 : #94, #121, #135, #139, #172, #180, #183, #255, #263, #278, #291, #298, #315, #317, #374, #408, #417, #470, #473, #486, #586, #652

Additional : #257 and #216 are duplicates of #52 #110 is not a duplicate and is actually invalid as it does not correctly identify the root cause and instead talks about out-of-gas due to loop which is impossible given the block gas limit of iota is 1B units as seen here https://explorer.evm.iota.org/block/424441.

#543 is definitely not a duplicate and actually invalid because the amount has been already scaled using “Constants.ACC_PRECISION_BITS” in the mentioned code.

Both set 1 and set 2 are still high severity issues.

sherlock-admin3

> Escalate
>
> This set of issues is actually two completely different sets of issues clubbed together incorrectly. One set is the “Voters will lose all bribe rewards forever if they do not claim their rewards after the last bribing period” ie. this issue #164 which emphasizes the loss of legit rewards `already earned by the voters`. In this case, the root cause and fix are different ie. voters will not be able to claim their rewards after certain time. In this case, the voters are affected.
>
> Here are duplicates of #164 : Set 1 : #145, #501, #495, Note that three issues #591, #161, #371 also identify the same issue but the description of these issues is technically incorrect because they say that the array out-of-bound revert will start happening after the lastVotingPeriod +1 which is not true because the array actually gets a length of lastID - startId + 2, as an extra empty element is added to the rewards array due to the wrong(but non-impactful) logic here. This means that while these three issues say revert will happen after lastId +1 period, the revert will actually only start happening after lastId + 2 periods as per the current

logic. Thus, these three issues should be deemed invalid due to the incorrect description of the issue, as they could not identify the correct conditions in which the bug manifests.

Thus, set 1 has only four valid issues : #164, #145, #501, #495

While the other set emphasises on the rewards that were not earned by any voter (maybe due to zero votes etc.) and thus they remain stuck in the briber contract. In this set, the root cause is the non-existence of a sweep function to restore rewards that were ‘’’not earned by anyone’. In this case, the briber itself is affected.

Solving one of them doesn’t solve the other as the fixes and root cause are completely unrelated.

Set 2 : #94, #121, #135, #139, #172, #180, #183, #255, #263, #278, #291, #298, #315, #317, #374, #408, #417, #470, #473, #486, #586, #652

Additional : #257 and #216 are duplicates of #52 #110 is not a duplicate and is actually invalid as it does not correctly identify the root cause and instead talks about out-of-gas due to loop which is impossible given the block gas limit of iota is 1B units as seen here https://explorer.evm.iota.org/block/424441.

#543 is definitely not a duplicate and actually invalid because the amount has been already scaled using “Constants.ACC_PRECISION_BITS” in the mentioned code.

Both set 1 and set 2 are still high severity issues.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Honour-d-dev

Escalate

This set of issues is actually two completely different sets of issues clubbed together incorrectly. One set is the “Voters will lose all bribe rewards forever if they do not claim their rewards after the last bribing period” ie. this issue #164 which emphasizes the loss of legit rewards `already earned by the voters`. In this case, the root cause and fix are different ie. voters will not be able to claim their rewards after certain time. In this case, the voters are affected.

Here are duplicates of #164 : Set 1 : #145, #501, #495, Note that three issues #591, #161, #371 also identify the same issue but the description

of these issues is technically incorrect because they say that the array out-of-bound revert will start happening after the lastVotingPeriod +1 which is not true because the array actually gets a length of lastID - startId + 2, as an extra empty element is added to the rewards array due to the wrong(but non-impactful) logic here. This means that while these three issues say revert will happen after lastId +1 period, the revert will actually only start happening after lastId + 2 periods as per the current logic. Thus, these three issues should be deemed invalid due to the incorrect description of the issue, as they could not identify the correct conditions in which the bug manifests.

Thus, set 1 has only four valid issues : #164, #145, #501, #495

While the other set emphasises on the rewards that were not earned by any voter (maybe due to zero votes etc.) and thus they remain stuck in the briber contract. In this set, the root cause is the non-existence of a sweep function to restore rewards that were ‘’’not earned by anyone’. In this case, the briber itself is affected.

Solving one of them doesn’t solve the other as the fixes and root cause are completely unrelated.

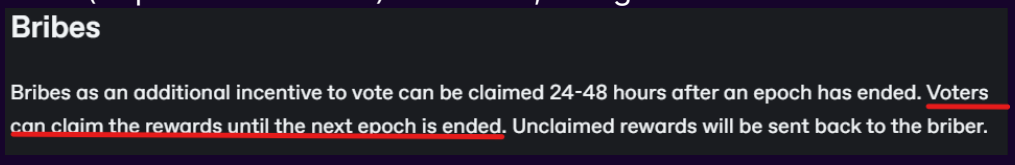
Set 2 : #94, #121, #135, #139, #172, #180, #183, #255, #263, #278, #291, #298, #315, #317, #374, #408, #417, #470, #473, #486, #586, #652

Additional : #257 and #216 are duplicates of #52 #110 is not a duplicate and is actually invalid as it does not correctly identify the root cause and instead talks about out-of-gas due to loop which is impossible given the block gas limit of iota is 1B units as seen here https://explorer.evm.iota.org/block/424441.

#543 is definitely not a duplicate and actually invalid because the amount has been already scaled using “Constants.ACC_PRECISION_BITS” in the mentioned code.

Both set 1 and set 2 are still high severity issues.

set 1 (duplicates of #164) is invalid , using the docs as a source of truth



rewards not claimed on the period after bribe ends should not be claimable by voters ,and should be left in contract to be swept by the briber. I believe the extra period empty period added in the `_rewards` array is for this purpose (ie to give the voters an extra period to claim their rewards)

0xSmartContract

Escalate ; 2 different set . (I do not aggre)

The reward mechanism in the BribeRewarder contract should be considered as a whole. Both users losing their earned rewards (#164) and unclaimed rewards getting stuck in the contract (#94) are different aspects of the same system. If these problems are solved separately, the integrity of the system can be compromised and unexpected interactions can occur.

Both problems are fundamentally due to deficiencies in the reward management logic of BribeRewarder. For example, the indexing logic in the _modify() function and the limitations of the claim() function contribute to both problems.

Changes made to solve one problem can affect the other. For example, a change made to ensure that users can always claim their rewards can also affect the management of unclaimed rewards

chinmay-farkya

> If these problems are solved separately, the integrity of the system can be compromised

I absolutely disagree that dupes are decided based on arriving at a common broader solution for two problems that are separate in the logic. That doesn't matter in issue duplication/separation on sherlock imo.

Honour-d-dev

@0xSmartContract set 1 of this escalation are actually invalid as mentioned in my previous comment. Voters are intentionally allowed only one extra period to claim their rewards. After which rewards are locked to be claimed by the briber.

If this is not the case then there will be conflict as to when the briber is allowed to sweep the contract for unclaimed rewards. hence the need for the deadline.

Slavchew

> > If these problems are solved separately, the integrity of the system can be compromised
>
> I absolutely disagree that dupes are decided based on arriving at a common broader solution for two problems that are separate in the logic. That doesn't matter in issue duplication/separation on sherlock imo.

I agree with this even more so that both issues have different fixes, locations, and impacts. The fact that both are associated with rewards does not make them one. If that were the case, there are over 5 award-related issues in the entire audit, should they be grouped together, NO of course.

WangSecurity

I've hidden one comment cause it seemed to be repeating the same point and not adding value.

About this issue, I agree that duplicates are: #145, #501, #495, #591.

#591 - says `Claiming rewards for any period after _lastVotingPeriod + 2 will be impossible` and not after `_lastVotingPeriod + 2` as said in the escalation comment, so I assume it's correct, excuse me if wrong, please correct. #371 and #161 agree to be incorrect.

But, @Honour-d-dev comment is very interesting, is there any evidence that this behaviour is not intended?

> Additional : https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/257 and https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/216 are duplicates of https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/52 https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/110 is not a duplicate and is actually invalid as it does not correctly identify the root cause and instead talks about out-of-gas due to loop which is impossible given the block gas limit of iota is 1B units as seen here https://explorer.evm.iota.org/block/424441. https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/543 is definitely not a duplicate and actually invalid because the amount has been already scaled using “Constants.ACC_PRECISION_BITS” in the mentioned code.

Completely agree with the part quoted above and plan to apply these changes.

Additionally, I agree that the second set mentioned in the escalation comment is a different issue. The root causes, fixes and vulnerability paths are different. Even though the impact is similar, it's not a ground to duplicate all of them into 1 issue. The best will be #172, the duplicates are: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/94, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/121, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/135, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/139, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/172, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/180, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/183, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/255, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/263, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/278, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/291, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/298, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/315, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/317, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/374, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/408, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/417,

https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/470, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/473, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/486, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/586, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/652

Waiting for an answer about lost rewards after 2 epochs before making the decision.

Slavchew

The @Honour-d-dev comment comes from just one piece of information in the docs, here's the full text:

https://docs.magicsea.finance/protocol/magic/magic-lum-voting#bribes

As far as I understand, this is just to say that the bribe rewards for the current period can be claimed after it ends.

"Voters can claim the rewards until the next epoch is ended."

The statement above - *rewards not claimed on the period after bribe ends should not be claimable by voters* - https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/164#issuecomment-2265233078 is not true as it's never states in the docs that bribe rewards can only be claimed 1 period **after the end** and no more, it states that *Voters can claim the rewards until the next epoch is ended*, **not the end one**.

Honour-d-dev

From my understanding **Until** means *"not after"* , no?

The statement in the docs: *"Voters can claim the rewards until the next epoch is ended"*

Is saying , "if the last bribe epoch for the `BribeRewarder` is `epoch-3`, then voters have **until** the end of `epoch-4` (i.e. the next epoch) to claim rewards"

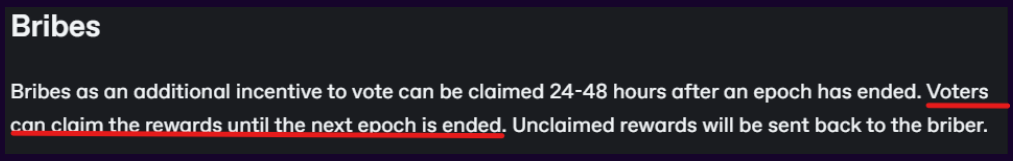
This makes even more sense ,considering the issue with *sweep functionality* so there needs to be a point where rewards not yet claimed can be considered as **forfeit** and the briber is allowed to sweep them.

@Slavchew I don't see how the full context of the docs disproves this

chinmay-farkya

Hey @WangSecurity this response is regarding the comment by honour-d-dev here : https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/164#issuecomment-2267602192

I believe that the docs, especially the relevant section, is outdated.



Here is the evidence for the docs being outdated.

1. The issue #346 says that "rewards can be claimed immediately after the epoch ends" which is non-impactful issue and thus invalid. The sponsor added a Won't fix label to that because the intention was not what the docs stated (the same section of docs relevant here), what makes sense is to allow reward claiming after the epoch, and that is what the code says.
2.

   Voters can claim the rewards until the next epoch is ended

If this statement in the docs was to be true then the claim function in BribeRewarder would have allowed only the one current finished period in the claim logic. See here.

The claim function forces to loop over all the finished periods(and not just the latest one) which means that the code was intended to allow staker to claim his rewards for all periods that have ended, and not just the one that ended latest.

So the code was really "meant to allow stakers to claim their rewards for all finished periods" otherwise the claim function would be having just the "getLatestFinishedPeriod()" in the claiming logic.

About #591 : yes, my bad it is a duplicate. But see that #145 is not a duplicate (this one says it will revert after lastVotingPeriod + 1 which is not true). I misnumbered the issue in the original escalation.

TLDR, #164 shall remain a separate valid high severity issue with duplicates : #501, #495, #591

Honour-d-dev

@chinmay-farkya

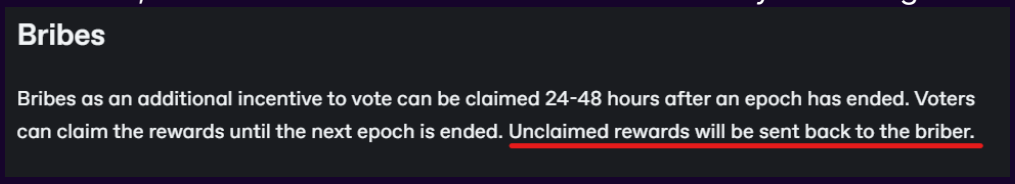
1. I do not agree that sponsor adding won't fix tag is evidence that the docs is outdated , same docs was provided for the audit as relevant protocol resources, no?. #346 just has no impact.
2.

   If this statement in the docs was to be true then the claim function in BribeRewarder would have allowed only the one current finished period in the claim logic. See here.

> The claim function forces to loop over all the finished periods(and not just the latest one) which means that the code was intended to allow staker to claim his rewards for all periods that have ended, and not just the one that ended latest.

I do not understand the argument here, if you're questioning why the bribe rewards for other periods (not the last one) are still claimable until the end of the last period+1 , then the answer is similar to #346, it's not impactful if the voter can still claim rewards for earlier periods as long as the `BribeRewarder` is still active.

It is **Impactful** ,however, if the voter can still claim rewards after the `BribeRewarder` is no longer active as this will interfere with the sweeping of unclaimed or undistributed tokens. If the voter is given unlimited time to claim rewards then at what point will it be safe to send back *unclaimed rewards* to the briber as stated by the docs, without the voters that has not claimed yet loosing their rewards?



chinmay-farkya

> I do not understand the argument here, if you're questioning why the bribe rewards for other periods (not the last one) are still claimable until the end of the last period+1 , then the answer is similar to https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/346, it's not impactful if the voter can still claim rewards for earlier periods as long as the BribeRewarder is still active.

I'm saying that if we were to believe what the docs say, then it should not have been encoded into the claim function to loop over a set of epochs. if "rewards could be claimed `only until` the next epoch ended" : If this would have been true and intended then the claim function would have the _modify call for only one epoch ie. "getLatestFinishedPeriod()" : ie. after epoch 1 ends, claim should call _modify for only 1st epoch and the loop would not have been in place at all. And that is not the case, so the docs do not portray what the code is designed for.

The current claim function logic means that the rewards claim has to be allowed for all finished periods uptil now, and not just the "only one latest finished period" {that is what the docs say}. Code takes precedence over the docs.

So in short, the claim has been designed to allow claiming all periods at once as per code (and code > docs), and rest you can read the original bug description.

> It is Impactful ,however, if the voter can still claim rewards after the BribeRewarder is no longer active as this will interfere with the sweeping of unclaimed or undistributed tokens.

In this context also, the docs are actually outdated/not updated acc to what the code is designed for currently. There was no sweep function in BribeRewarder at all. Sweeping out rewards earned by users also does not make sense, they should be able to claim them if they have been accrued. The bribeRewarder can only be used once, so the unclaimed funds can sit without any problems.

The current recommendation around "briberewarder should have a sweep function" pointed out in other bug is for "undistributed rewards" and not for "unclaimed rewards", because sweeping out funds which were not even accrued to anybody due to no votes in an apoch(for ex.) makes sense, but sweeping out rewards earned by users and still sitting in the briberewarder as unclaimed does not make sense.

> If the voter is given unlimited time to claim rewards then at what point will it be safe to send back unclaimed rewards to the briber

They should never be sent back, only the undistributed rewards shall be. That can be easily done by recording all accrued rewards in a separate variable lets say "Rewardsdistributed" and only sweeping out balance - rewardsdistributed back to the protocol. Also see thats how it is done in the BaseRewarder here : maintaining a reserve variable.

Slavchew

@WangSecurity Instead of trying to descramble the words in the documentation, can't the sponsor just come and write if they think it's okay for users to lose bribe rewards if they haven't taken them max 2 periods after that? Everyone will agree that it is not acceptable, because the rewards are not sent and everyone has to claim them and not everyone would have the opportunity to claim them during this period.

I think that will settle the whole escalation.

Honour-d-dev

> I'm saying that if we were to believe what the docs say, then it should not have been encoded into the claim function to loop over a set of epochs. if "rewards could be claimed only until the next epoch ended" : If this would have been true and intended then the claim function would have the _modify call for only one epoch ie. "getLatestFinishedPeriod()" : ie. after epoch 1 ends, claim should call _modify for only 1st epoch and the loop would not have been in place at all. And that is not the case, so the docs do not portray what the code is designed for.
>
> So in short, the claim has been designed to allow claiming all periods at once as per code (and code > docs), and rest you can read the original bug description.

Looping over all epochs to calculate rewards is a design decision (as opposed to having the voter specify the exact epoch they voted/have rewards in) it does not

contradict or invalidate the docs.

> In this context also, the docs are actually outdated/not updated acc to what the code is designed for currently. There was no sweep function in BribeRewarder at all. Sweeping out rewards earned by users also does not make sense, they should be able to claim them if they have been accrued. The bribeRewarder can only be used once, so the unclaimed funds can sit without any problems.

Again, docs is not outdated , `BribeRewarder` can be used multiple times actually (*"The bribeRewarder can only be used once"* is false) and sweeping unclaimed rewards makes sense as it is in the protocol docs(which makes it public information) so voters would be aware of it and know that they have a 14-day window to claim rewards

> They should never be sent back, only the undistributed rewards shall be. That can be easily done by recording all accrued rewards in a separate variable lets say "Rewardsdistributed" and only sweeping out balance - rewardsdistributed back to the protocol. Also see thats how it is done in the BaseRewarder here : maintaining a reserve variable.

It's not possible to record all the rewards accrued by users in this implementation of the `masterChef` algo because rewards still accrue(or is distributed) even when nobody votes(as distribution is time-based i.e. `emissionsPerSecond`, to encourage voters to vote as early as possible) and a voter's specific reward is dependent on the **time** they voted and the total votes in that epoch. The `_reserve` variable in BaseRewarder is actually the total rewards available in the contract and the `_totalUnclaimedRewards` works a differently there because there's no concept of epochs that reset the total supply to 0, just a start and end time. Hence why Briberewarder is not BaseRewarder

chinmay-farkya

> Looping over all epochs to calculate rewards is a design decision (as opposed to having the voter specify the exact epoch they voted/have rewards in) it does not contradict or invalidate the docs.

That is completely false. That does invalidate the docs. The docs say that only the latest period shall be allowed to be claimed, but the code loops over all periods including the latest one which means it is meant to allow claiming for all past periods, how are these two equivalent statements ? They are clearly not. So the code needs to be believed and the code has a bug ie. #164

> BribeRewarder can be used multiple times actually ("The bribeRewarder can only be used once" is false)

Again that is completely false. here is a surprise for you : the briberewarder can only be initialized once : see https://github.com/sherlock-audit/2024-06-magicsea/

blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/rewarders/BribeRewarder.sol#L227

> It's not possible to record all the rewards accrued by users in this implementation of the masterChef algo because rewards still accrue(or is distributed) even when nobody votes

Again completely false the rewards would get stuck if no one voted thats what the other set of issues in another bug is. Please read #172

lets leave baserewarder out of the current discussion an focus on the current bug.

Honour-d-dev

> The docs say that only the latest period shall be allowed to be claimed,

The docs does not say or imply this please. This is a false statement

> Again that is completely false. here is a surprise for you : the briberewarder can only be initialized once

In original comment https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/164#issuecomment-2284552877 you said **used** not initialized,

> The bribeRewarder can only be used once, so the unclaimed funds can sit without any problems.

i interpreted as **used** to claim rewards (which can be done multiple times)

> Again completely false the rewards would get stuck if no one voted thats what the other set of issues in another bug is.

I did not say rewards don't get stuck, i said rewards are still accrued if no one votes. For example if the BribeRewarder is to distribute 100usd to voters for that epoch, if the first voter votes after 7 days(half-way into the epoch) then they can only earn a maximum of 50usd assuming no one else votes on the bribed pool. Which means rewards still accrue regardless of votes or not (but to nobody if there's no votes). i can provide POC of this if you're still in doubt.

WangSecurity

> Instead of trying to descramble the words in the documentation, can't the sponsor just come and write if they think it's okay for users to lose bribe rewards if they haven't taken them max 2 periods after that? Everyone will agree that it is not acceptable, because the rewards are not sent and everyone has to claim them and not everyone would have the opportunity to claim them during this period

I would like to do that, but the sponsor is OOS for 2 weeks, so we cannot get this information.

But I believe we have sufficient arguments to settle the discussion. The docs say:

> Voters can claim the rewards until the next epoch is ended.

In this case, if the last epoch is 5 (arbitrary number), then we shouldn't be able to claim rewards after the 6th epoch has ended. But, as I understand and as this report shows, the rewards can be claimed after the 6th epoch has ended, but cannot be claimed after the 7th epoch. If I'm correct here, then we have contextual evidence that the docs are outdated.

Then the question about not having the sweep function for the briber to get back the rewards. In that sense, it's still a valid bug, since not all rewards can be distributed, so they should be swept by the briber.

Hope that answers the questions and I believe it's a fair decision here. About #145, agree it shouldn't be a duplicate. Hence, my decision is to accept the escalation, apply the changes expressed here, except #145 being invalid. Both families will have High severity.

iamnmt

@WangSecurity @chinmay-farkya

I believe #145 stating the correct condition in which the bug occurs

> when `Voter#getLatestFinishedPeriod()` is greater than `BribeRewarder#_lastVotingPeriod + 1`

To elaborate on this, `Voter#getLatestFinishedPeriod()` is greater than `BribeRewarder#_lastVotingPeriod + 1` if and only if (`_currentVotingPeriodId` is equal to `BribeRewarder#_lastVotingPeriod + 2` and current voting period is ended) or `_currentVotingPeriodId` is greater than `BribeRewarder#_lastVotingPeriod + 2`

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/Voter.sol#L471

I do not see anything wrong with this condition. Please correct me if I am wrong.

This condition is the same as "after 2 periods have passed after the last bribing period".

> But see that https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/145 is not a duplicate (this one says it will revert after lastVotingPeriod + 1 which is not true)

Nowhere in #145 mention that it will revert after lastVotingPeriod + 1.

WangSecurity

> Users can not claim from BribeRewarder when Voter#getLatestFinishedPeriod() is greater than BribeRewarder#_lastVotingPeriod + 1, which will result in DoS on BribeRewarder#claim and loss of funds

Here's the impact from #145. It says the rewards will be unable to be claimed after `_lastVotingPeriod + 1`. The rewards will be unable to be claimed because the function will revert. But, that's not fully correct, it's still possible to claim the rewards after `_lastVotingPeriod + 1` during `_lastVotingPeriod + 2`. That's why I believe the report is not sufficient because rewards become unable to be claimed after `_lastVotingPeriod + 2` when the report says `+1`. Hence, I believe it's incorrect.

The decision remains as expressed here

iamnmt

@WangSecurity @chinmay-farkya

> it's still possible to claim the rewards after `_lastVotingPeriod + 1` during `_lastVotingPeriod + 2`

You are correct.

> It says the rewards will be unable to be claimed after _lastVotingPeriod + 1

No, it is not. #145 intentionally mentions the function `Voter#getLatestFinishedPeriod()`

> `Voter#getLatestFinishedPeriod()` is greater than `BribeRewarder#_lastVotingPeriod + 1`

If the current voting period is `_lastVotingPeriod + 2` and the voting period is not over, then `Voter#getLatestFinishedPeriod()` will return `_lastVotingPeriod + 1`. If the current voting period is `_lastVotingPeriod + 2` and the voting period is over, then `Voter#getLatestFinishedPeriod()` will return `_lastVotingPeriod + 2`. That is why I believe this condition is the same as "after 2 periods have passed after the last bribing period".

Another way to look at this condition is just looking at `BribeRewarder#claim` in isolation. There are `_lastVotingPeriod - _startVotingPeriod + 2` elements in the `_rewards` array, meaning the array can be indexed up to `_lastVotingPeriod + 1`. Indexing an element greater than `_lastVotingPeriod + 1` will cause revert. When `BribeRewarder#claim` is called, the `Voter#getLatestFinishedPeriod()`th element will be indexed. Thus, if `Voter#getLatestFinishedPeriod()` is greater than `_lastVotingPeriod + 1`, then the function call will revert.

Because of the reasons above, I believe the condition in #145 is correct. I kindly request you to review your decision again.

chinmay-farkya

Yeah I relooked at it, and now I see you are right.

#145 is a correct duplicate of this. "getLatestFinishedPeriod > lastVotingPeriod + 1" essentially means the same thing as "last finsihed period = lastVotingPeriod + 2"

Sorry got confused earlier with the wording.

WangSecurity

Excuse me, @iamnmt is correct and I didn't notice this. Planning to accept the escalation and apply the changes as here making 2 families with High severity.

PeterSRWeb3

@WangSecurity @0xSmartContract This issue should be a duplicate https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/409 of the second family of issues, because it's pointing out the same problem and it's proposing the same solution. Thanks in advance!

WangSecurity

As I understand #409 report is incorrect. The example given is that the claiming of rewards will revert when the latest finished voting period is 5 if the rewards are distributed from 2 to 4 periods. But, as I understand this is incorrect and the rewards can still be claimed until 6th voting period is finished, based on the discussion above and this report.

Simply put, the #409 report says the claiming of rewards is impossible after `_latestVotingPeriod + 1`, when in fact it's still possible and reverts when `_latestVotingPeriod + 2`. Correct me if my understanding of #409 is incorrect.

The decision remains as expressed here

PeterSRWeb3

@WangSecurity Thanks sir for taking a look! This is due to the fact that I've ran the test using the first reccomendation. Of course this is my mistake, but the report is valid for me, because it's pointing the same issue and it's proposing the same solution. Please take a look again sir. I totally understand you and I will do my best, to not make this type of mistakes again.

WangSecurity

Still, the report is incorrect, hence, my decision remains the same.

PeterSRWeb3

Okay, sir, my bad in this situation. Thanks for the effort

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- chinmay-farkya: accepted

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-5: Voting does not take into account end of staking lock period

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/166

Found by

0xAnmol, 0xpranav, Aymen0909, BengalCatBalu, ChinmayF, DPS, Hunter, KupiaSec, LeFy, PUSH0, Reentrants, Silvermist, WildSniper, anonymousjoe, araj, aslanbek, blackhole, blockchain555, cocacola, coffiasd, dany.armstrong90, dhank, dimulski, iamnmt, jennifer37, jsmi, kmXAdam, novaman33, qmdddd, radin200, rsam_eth, santipu_, scammed, snapishere, tedox, utsav, web3pwn, ydlee, zarkk01

Summary

The protocol allows to vote in `Voter` contract by means of staked position in `MlumStaking`. To vote, user must have staking position with certain properties. However, the voting does not implement check against invariant that the remaining lock period needs to be longer then the epoch time to be eligible for voting. Thus, it is possible to vote with stale voting position. Additionally, if position's lock period finishes inside of the voting epoch it is possible to vote, withdraw staked position, stake and vote again in the same epoch. Thus, voting twice with the same stake amount is possible from time to time. Ultimately, the invariant that voting once with same balance is only allowed is broken as well. The voting will decide which pools receive LUM emissions and how much.

Vulnerability Detail

The documentation states that:

> Who is allowed to vote Only valid Magic LUM Staking Position are allowed to vote. The overall lock needs to be longer then 90 days and the remaining lock period needs to be longer then the epoch time.

User who staked position in the `MlumStaking` contract gets NFT minted as a proof of stake with properties describing this stake. Then, user can use that staking position to vote for pools by means of `vote()` in `Voter` contract. The `vote()` functions checks if `initialLockDuration` is higher than `_minimumLockTime` and `lockDuration` is higher than `_periodDuration` to process further. However, it does not check whether the remaining lock period is longer than the epoch time. Thus, it is possible to vote with stale staking position. Also, current implementation makes `renewLockPosition` and `extendLockPosition` functions useless.

```
function vote(uint256 tokenId, address[] calldata pools, uint256[] calldata
↪  deltaAmounts) external {
```


SHERLOCK

```
    if (pools.length != deltaAmounts.length) revert IVoter__InvalidLength();

    // check voting started
    if (!_votingStarted()) revert IVoter_VotingPeriodNotStarted();
    if (_votingEnded()) revert IVoter_VotingPeriodEnded();

    // check ownership of tokenId
    if (_mlumStaking.ownerOf(tokenId) != msg.sender) {
        revert IVoter__NotOwner();
    }

    uint256 currentPeriodId = _currentVotingPeriodId;
    // check if alreay voted
    if (_hasVotedInPeriod[currentPeriodId][tokenId]) {
        revert IVoter__AlreadyVoted();
    }

    // check if _minimumLockTime >= initialLockDuration and it is locked
    if (_mlumStaking.getStakingPosition(tokenId).initialLockDuration <
↪   _minimumLockTime) {
        revert IVoter__InsufficientLockTime();
    }
    if (_mlumStaking.getStakingPosition(tokenId).lockDuration < _periodDuration)
↪   {
        revert IVoter__InsufficientLockTime();
    }

    uint256 votingPower =
↪   _mlumStaking.getStakingPosition(tokenId).amountWithMultiplier;

    // check if deltaAmounts > votingPower
    uint256 totalUserVotes;
    for (uint256 i = 0; i < pools.length; ++i) {
        totalUserVotes += deltaAmounts[i];
    }

    if (totalUserVotes > votingPower) {
        revert IVoter__InsufficientVotingPower();
    }

    IVoterPoolValidator validator = _poolValidator;

    for (uint256 i = 0; i < pools.length; ++i) {
        address pool = pools[i];

        if (address(validator) != address(0) && !validator.isValid(pool)) {
            revert Voter__PoolNotVotable();
```


SHERLOCK

```
        }

        uint256 deltaAmount = deltaAmounts[i];

        _userVotes[tokenId][pool] += deltaAmount;
        _poolVotesPerPeriod[currentPeriodId][pool] += deltaAmount;

        if (_votes.contains(pool)) {
            _votes.set(pool, _votes.get(pool) + deltaAmount);
        } else {
            _votes.set(pool, deltaAmount);
        }

        _notifyBribes(_currentVotingPeriodId, pool, tokenId, deltaAmount); //
↪   msg.sender, deltaAmount);
    }

    _totalVotes += totalUserVotes;

    _hasVotedInPeriod[currentPeriodId][tokenId] = true;

    emit Voted(tokenId, currentPeriodId, pools, deltaAmounts);
}
```

The documentation states that minimum lock period for staking to be eligible for voting is 90 days. The documentation states that voting for pools occurs biweekly.

Thus, assuming the implementation with configuration presented in the documentation, every 90 days it is possible to vote twice within the same voting epoch by:

- voting,
- withdrawing staked amount,
- creating new position with staking token,
- voting again.

```
function createPosition(uint256 amount, uint256 lockDuration) external override
↪   nonReentrant {
    // no new lock can be set if the pool has been unlocked
    if (isUnlocked()) {
        require(lockDuration == 0, "locks disabled");
    }

    _updatePool();
```

```
    // handle tokens with transfer tax
    amount = _transferSupportingFeeOnTransfer(stakedToken, msg.sender, amount);
    require(amount != 0, "zero amount"); // createPosition: amount cannot be null

    // mint NFT position token
    uint256 currentTokenId = _mintNextTokenId(msg.sender);

    // calculate bonuses
    uint256 lockMultiplier = getMultiplierByLockDuration(lockDuration);
    uint256 amountWithMultiplier = amount * (lockMultiplier + 1e4) / 1e4;

    // create position
    _stakingPositions[currentTokenId] = StakingPosition({
        initialLockDuration: lockDuration,
        amount: amount,
        rewardDebt: amountWithMultiplier * (_accRewardsPerShare) /
↪   (PRECISION_FACTOR),
        lockDuration: lockDuration,
        startLockTime: _currentBlockTimestamp(),
        lockMultiplier: lockMultiplier,
        amountWithMultiplier: amountWithMultiplier,
        totalMultiplier: lockMultiplier
    });

    // update total lp supply
    _stakedSupply = _stakedSupply + amount;
    _stakedSupplyWithMultiplier = _stakedSupplyWithMultiplier +
↪   amountWithMultiplier;

    emit CreatePosition(currentTokenId, amount, lockDuration);
}
```

Proof of Concept

Scenario 1:

```
function testGT_vote_twice_with_the_same_stake() public {
        vm.prank(DEV);
        _voter.updateMinimumLockTime(2 weeks);

        _stakingToken.mint(ALICE, 1 ether);

        vm.startPrank(ALICE);
        _stakingToken.approve(address(_pool), 1 ether);
        _pool.createPosition(1 ether, 2 weeks);
        vm.stopPrank();
```


SHERLOCK

```
        skip(1 weeks);

        vm.prank(DEV);
        _voter.startNewVotingPeriod();

        vm.startPrank(ALICE);
        _voter.vote(1, _getDummyPools(), _getDeltaAmounts());
        vm.expectRevert(IVoter.IVoter__AlreadyVoted.selector);
        _voter.vote(1, _getDummyPools(), _getDeltaAmounts());
        vm.stopPrank();

        assertEq(_voter.getTotalVotes(), 1 ether);

        skip(1 weeks + 1);

        vm.startPrank(ALICE);
        _pool.withdrawFromPosition(1, 1 ether);
        vm.stopPrank();

        vm.startPrank(ALICE);
        vm.expectRevert();
        _voter.vote(1, _getDummyPools(), _getDeltaAmounts());
        _stakingToken.approve(address(_pool), 1 ether);
        _pool.createPosition(1 ether, 2 weeks);
        _voter.vote(2, _getDummyPools(), _getDeltaAmounts());
        vm.stopPrank();

        assertEq(_voter.getTotalVotes(), 2 ether);
    }
```

Scenario 2:

```
function testGT_vote_twice_with_the_same_stake() public {
        vm.prank(DEV);
        _voter.updateMinimumLockTime(2 weeks);

        _stakingToken.mint(ALICE, 1 ether);

        vm.startPrank(ALICE);
        _stakingToken.approve(address(_pool), 1 ether);
        _pool.createPosition(1 ether, 2 weeks);
        vm.stopPrank();

        skip(1 weeks);

        vm.prank(DEV);
```

```
        _voter.startNewVotingPeriod();

        vm.startPrank(ALICE);
        _voter.vote(1, _getDummyPools(), _getDeltaAmounts());
        vm.expectRevert(IVoter.IVoter__AlreadyVoted.selector);
        _voter.vote(1, _getDummyPools(), _getDeltaAmounts());
        vm.stopPrank();

        assertEq(_voter.getTotalVotes(), 1 ether);

        skip(1 weeks + 1);

        vm.startPrank(ALICE);
        _pool.withdrawFromPosition(1, 1 ether);
        vm.stopPrank();

        vm.startPrank(ALICE);
        vm.expectRevert();
        _voter.vote(1, _getDummyPools(), _getDeltaAmounts());
        _stakingToken.approve(address(_pool), 1 ether);
        _pool.createPosition(1 ether, 2 weeks);
        _voter.vote(2, _getDummyPools(), _getDeltaAmounts());
        vm.stopPrank();

        assertEq(_voter.getTotalVotes(), 2 ether);
    }
```

Impact

A user can vote with stale staking position, then withdraw the staking position with any consequences. Additionally, a user can periodically vote twice with the same balance of staking tokens for the same pool to increase unfairly the chance of the pool being selected for further processing.

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/Voter.sol#L172 https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/Voter.sol#L175

Tool used

Manual Review

Recommendation

It is recommended to enforce the invariant that the remaining lock period must be longer than the epoch time to be eligible for voting. Additionally, It is recommended to prevent double voting at any time. One of the solution can be to prevent voting within the epoch if staking position was not created before epoch started.

Discussion

0xSmartContract

The vulnerability identified involves the voting mechanism within the `Voter` contract that allows users to vote using staked positions in the `MlumStaking` contract.

The issue arises because the contract does not validate if the remaining lock period of a staking position is longer than the voting epoch. As a result, users can vote with a staking position that has a lock period ending within the voting epoch, allowing potential double voting by.

This vulnerability can lead to:

- Voting with stale staking positions.
- Double voting within the same epoch, thus skewing the vote results unfairly.
- Breaking the invariant that each stake amount should only vote once per epoch.

The `vote` function currently checks:

1. The ownership of the tokenId.
2. If the user has already voted in the current period.
3. The initial lock duration and current lock duration against minimum thresholds.

However, it fails to ensure that the remaining lock period exceeds the epoch time, allowing potential manipulation as described.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/25

chinmay-farkya

Escalate

Some issues have been incorrectly duped with this issue :

#219 is not a duplicate and is actually invalid as it involves a trusted admin to use a function at a wrong time. The assertion that the admin will start a new voting period when one is already running is completely unreasonable. Also, the readme

clearly mentions that a keeper bot is meant to start new epochs periodically and its off-chain and out-of-scope. #306 is not a duplicate and is invalid as it ties to the same reasoning for #219 above #504 is again not a duplicate as it ties to the same reasoning for #219 above #405 is again not a duplicate as it ties to the same reasoning for #219 above #407 is not a duplicate and is invalid due to the same reason as #219, cited above.

#252 is not a duplicate of this and is invalid/low because it just points out a difference between docs and code but it has no real impact or loss of funds. It talks about the ability to claim rewards later after more epoch periods have passed, which is completely normal and does not relate to a malicious action or a loss to the staker. #413 is invalid because frontrunning is not possible on iota #419 is a completely different issue and is a low because its an admin controlled action. Admin can wait for the current voting period to end, then update minimumLockTime and then start a new voting period. Since admin is trusted, they are expected to wait for current period to end and only checking time config starting from the next period. From sherlock docs

> An admin action can break certain assumptions about the functioning of the code. Example: Pausing a collateral causes some users to be unfairly liquidated or any other action causing loss of funds. This is not considered a valid issue.

There is a separate set of issues described as “expired positions can keep harvesting rewards”. This is not the same as what this issue 166 describes. Fixing 166 by not allowing voting using already expired or to-be-expired-soon during the voting period is not going to fix this set of issues, because harvesting rewards will still be possible in MLUMStaking.sol.

This set includes: #6, #530, #362 and #253

This set should be made a separate medium issue. It requires expired locks to be kicked or harvesting to be disabled for already expired locks.

sherlock-admin3

> Escalate
>
> Some issues have been incorrectly duped with this issue :
>
> #219 is not a duplicate and is actually invalid as it involves a trusted admin to use a function at a wrong time. The assertion that the admin will start a new voting period when one is already running is completely unreasonable. Also, the readme clearly mentions that a keeper bot is meant to start new epochs periodically and its off-chain and out-of-scope. #306 is not a duplicate and is invalid as it ties to the same reasoning for #219 above #504 is again not a duplicate as it ties to the same reasoning for #219 above #405 is again not a duplicate as it ties to

the same reasoning for #219 above #407 is not a duplicate and is invalid due to the same reason as #219, cited above.

#252 is not a duplicate of this and is invalid/low because it just points out a difference between docs and code but it has no real impact or loss of funds. It talks about the ability to claim rewards later after more epoch periods have passed, which is completely normal and does not relate to a malicious action or a loss to the staker. #413 is invalid because frontrunning is not possible on iota #419 is a completely different issue and is a low because its an admin controlled action. Admin can wait for the current voting period to end, then update minimumLockTime and then start a new voting period. Since admin is trusted, they are expected to wait for current period to end and only checking time config starting from the next period. From sherlock docs

> An admin action can break certain assumptions about the functioning of the code. Example: Pausing a collateral causes some users to be unfairly liquidated or any other action causing loss of funds. This is not considered a valid issue.

There is a separate set of issues described as “expired positions can keep harvesting rewards”. This is not the same as what this issue 166 describes. Fixing 166 by not allowing voting using already expired or to-be-expired-soon during the voting period is not going to fix this set of issues, because harvesting rewards will still be possible in MLUMStaking.sol.

This set includes: #6, #530, #362 and #253

This set should be made a separate medium issue. It requires expired locks to be kicked or harvesting to be disabled for already expired locks.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xSmartContract

#219, #306, #504, #405, #407 , #419; This group is considered invalid under "Admin Input and responsibility", I agree.

#252 low/Invalid , I agree (There is an escalation process in 252, so 252 should be evaluated as low/invalid.)

#413 I do not agree.(The verdict is that front-running on IOTA is valid) This issue was discussed and confirmed with Sherlock officials; The verdict is that front-running on IOTA is valid, but can be at most medium severity, since it's not

reliable and uncontrollable by the attacker, though they can mitigate randomness with sending lot's of transactions. But later this year, this will be invalid, cause when IOTA launches version 2.0, they will have no mempool at all, while the current one is public.

#6, #530, #362, #253 Set includes . I do not agree

0xf1b0

> #413 is invalid because frontrunning is not possible on iota

Front running becomes possible when the reward schedule is known in advance. There's no necessity to monitor a pool and send transactions within the same block; this can be accomplished an hour earlier.

Although it may not be the most suitable title, the issue remains definitively valid.

J4X-98

> #6, #530, #362, #253 Set includes . I do not agree

@0xSmartContract what do you mean by that?

Additionally regarding #413, tx are randomly ordered on the IOTA evm which makes frontrunning impossible. The IOTA EVM documentation also states this:

"To protect against Maximal Extractable Value attacks, the IOTA Smart Contract chain natively randomizes transaction ordering, making it impossible to determine the content of a block in order to manipulate it."

0xSmartContract

> #413 I do not agree.(The verdict is that front-running on IOTA is valid)
> This issue was discussed and confirmed with Sherlock officials; The verdict is that front-running on IOTA is valid, but can be at most medium severity, since it's not reliable and uncontrollable by the attacker, though they can mitigate randomness with sending lot's of transactions. But later this year, this will be invalid, cause when IOTA launches version 2.0, they will have no mempool at all, while the current one is public.

Front-running on IOTA is valid I also said that there was no front-run like you, but Sherlock confirmed it , @WangSecurity ;

This issue was discussed and confirmed with Sherlock officials; The verdict is that front-running on IOTA is valid, but can be at most medium severity, since it's not reliable and uncontrollable by the attacker, though they can mitigate randomness with sending lot's of transactions. But later this year, this will be invalid, cause when IOTA launches version 2.0, they will have no mempool at all, while the current one is public.

J4X-98

@0xSmartContract,

If the head of judging is of that opinion I guess we'll have to accept it. Although it makes almost no sense on a sandwiching attack, as the user would only be able to extract a minor part of the funds as he has to split his capital used for the sandwich into many txs.

Nevertheless could you tell me what you meant by the quoted part on top of my message?

0xSmartContract

> @0xSmartContract,
>
> If the head of judging is of that opinion I guess we'll have to accept it. Although it makes almost no sense on a sandwiching attack, as the user would only be able to extract a minor part of the funds as he has to split his capital used for the sandwich into many txs.
>
> Nevertheless could you tell me what you meant by the quoted part on top of my message?

A set was mentioned in Escalate, I just wanted to say that this issue should not be separated into a different set.

J4X-98

@0xSmartContract,

What's your reasoning for that? As mentioned by me on the escalation of #6 these are completely different issues.

0xSmartContract

> @0xSmartContract,
>
> What's your reasoning for that? As mentioned by me on the escalation of #6 these are completely different issues.

#6 and #166

- Both issues focus on gaining unfair advantage through positions whose lock period has expired.
- Both issues deal with the ability of positions whose lock period has expired to remain valid in the current system.
- Both issues propose that positions should be revoked if their lock period has expired to prevent this situation.

chinmay-farkya

I disagree with your take that these are the same issues. Just because they happen to be in the same component of functionality, they can't be duped.

One problem exists in the voter contract, and another exists in MLUMStaking. In 166, we need to check for remaining lock duration which can also expire during the period itself.

Thus the solution for 6 which is to kick expired positions is not going to solve 166 as positions that have not expired yet but soon-to-expire will still be able to take advantage in voting. They have separate problems and solutions, and are not dups.

> Both issues focus on gaining unfair advantage through positions whose lock period has expired.

Not entirely true. 166 is also about soon-to-expire positions (ie. anytime in the next 14 days) and they can't be kicked.

> Both issues deal with the ability of positions whose lock period has expired to remain valid in the current system.

Again not true due to the involvement of soon-to-expire positions

> Both issues propose that positions should be revoked if their lock period has expired to prevent this situation.

Now that is completely false. 166 proposes to check remaining duration and not to "revoke positions" as some positions might still be active and would expire during the period itself, they can't be simply kicked.

J4X-98

- Both issues focus on gaining unfair advantage through positions whose lock period has expired.

This is a similar impact. But this is no reason for grouping issues. If you find 10 different ways to drain funds in a contract with different root causes you also can't merge them into one issue just because they all have the same impact.

- Both issues deal with the ability of positions whose lock period has expired to remain valid in the current system.

This is a similar scenario, but still no reason for duping. That would be the same if you had multiple issues that occur while a contract is paused, but all originate from different root causes. You also can't duplicate those together.

- Both issues propose that positions should be revoked if their lock period has expired to prevent this situation.

This is not true. #166 does not state anything about revoking the positions once they are expired. It just fixes the if statement.

WangSecurity

About the escalation comment:

1. #219, #306, #504, #405, #407 -- agree to invalidate based on the reasoning in the escalation comment.
2. 252 is already considered on its escalation.
3. #413 -- As I understand, the front-running is still possible (until the launch of IOTA V2 with no mempool), but unreliable. But, the report doesn't have sufficient proof of the issue considering there are other constraints of the lock duration, for example. Hence, I agree it should be invalid.
4. #419 -- agree with the escalation comment, I believe it's precise.
5. I agree that issues #6, #530, #362 and #253 are a different family. #6 is escalated, so the discussion about them will be continued under #6.

Planning to accept the escalation and apply the changes above.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- chinmay-farkya: accepted

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-6: Funds unutilized for rewards may get stranded in BribeRewarder

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/172

Found by

0uts1der, Aymen0909, ChinmayF, Honour, KupiaSec, PUSH0, PeterSR, Reentrants, Silvermist, araj, aslanbek, coffiasd, dany.armstrong90, dimulski, iamnmt, novaman33, pashap9990, rsam_eth, scammed, slowfi, utsav, web3pwn

Summary

When a bribe provider starts the bribe cycle on a `BribeRewarder.sol` contract, they are required to fund the contract with the total amount of rewardToken required across all the bribePeriods (enforced by the check here), which is amountPerPeriod x number of bribe periods they want to incentivize.

However if any of these funds remain untilized for reward distribution, they will remain stranded in the contract forever as there is no way to recover tokens.

Vulnerability Detail

The bribe cycle can be started on a BribeRewarder contract by calling `fundAndBribe()` or `bribe()`, where it is checked that the balance of the rewardToken in this contract is at least equal to the amount required to distribute across all defined bribe periods.

total amount required = amountPerPeriod x number of bribe periods briber wants to incentivize.

Lets say X amount of funds is required and the contract has been funded with exactly X. Now if in any case, the funds do not get utilized, they will remain stuck in the contract forever : as there is no way to sweep these tokens out.

This is possible due to the following reasons :

(1). The full `amountPerPeriod` will be utilized for distribution even if there is only one voter for the associated poolID in a given period. But if there is no voter, these funds will not be utilized for any other periods too. A pool might not get any voter under normal operations :

- The pool is not in the top pool IDs set on the Voter contract, thus not earning any rewards on the MasterChef, so users might want to vote for the top pools to potentially earn more LUM from there.

- The associated pool might get hacked, or drained, or emptied by LPs : thus no user would want to vote for such a pool after this happens, instead they would go for better incentives on LP swap fees.

(2). Some amount of tokens might remain unutilized even under normal circumstances and actively running bribing periods due to rounding in calculations. An instance of this is possible when calculating total rewards accrued in `BribeRewarder._modify => _calculateRewards` :

```
function _calculateRewards(uint256 periodId) internal view returns (uint256) {
    (uint256 startTime, uint256 endTime) =
↪   IVoter(_caller).getPeriodStartEndtime(periodId);

    if (endTime == 0 || startTime > block.timestamp) {
        return 0;
    }

    uint256 duration = endTime - startTime;
    uint256 emissionsPerSecond = _amountPerPeriod / duration;

    uint256 lastUpdateTimestamp = _lastUpdateTimestamp;
    uint256 timestamp = block.timestamp > endTime ? endTime : block.timestamp;
    return timestamp > lastUpdateTimestamp ? (timestamp - lastUpdateTimestamp) *
↪   emissionsPerSecond : 0;
}
```

Here `emissionsPerSecond = _amountPerPeriod / duration` : this could round down leading to some portion of the funds unutilized in the rewards calculations, and will be left out of the rewards for whole of the bribing cycle.

Impact

Due to the two reasons mentioned above, it is possible that some amount of rewardToken remains stranded in the contract, but there is no way for the bribe provider to recover these tokens.

These funds will be lost permanently, especially in the first case above, the amount would be large.

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/rewarders/BribeRewarder.sol#L308

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/rewarders/BaseRewarder.sol#L208

Tool used

Manual Review

Recommendation

BribeRewarder.sol should have a method to recover unutilized reward tokens, just like the sweep function in BaseRewarder.sol. This will prevent the permanent loss of funds for the briber.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/36

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-7: Attacker can block all votes to a specific pool by triggering an overflow error

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/237

Found by

santipu_

Summary

An attacker can deploy a `BribeRewarder` contract using a custom token and distribute a huge number of those tokens to trigger an overflow error and prevent users from voting in a specific pool.

Vulnerability Detail

In the MagicSea protocol, bribing is a core mechanism. In short, users can permissionlessly create `BribeRewarder` contracts and fund them with tokens to bribe users in exchange for voting in specific pools.

When users vote for a specific pool through calling `Voter::vote`, the function `_notifyBribes` is called, which calls each `BribeRewarder` contract that is attached to that voted pool:

```
function _notifyBribes(uint256 periodId, address pool, uint256 tokenId, uint256
↪  deltaAmount) private {
    IBribeRewarder[] storage rewarders = _bribesPerPriod[periodId][pool];
    for (uint256 i = 0; i < rewarders.length; ++i) {
        if (address(rewarders[i]) != address(0)) {
>>>         rewarders[i].deposit(periodId, tokenId, deltaAmount);
            _userBribesPerPeriod[periodId][tokenId].push(rewarders[i]);
        }
    }
}
```

An attacker can use this bribing mechanism to create malicious `BribeRewarder` contracts that will revert the transaction each time a user tries to vote for a specific pool. The attack path is the following:

1. An attacker creates a custom token and mints the maximum amount of tokens (`type(uint256).max`).
2. The attacker creates a `BribeRewarder` contract using the custom token.

3. The attacker transfers the total supply of the token to the rewarder contracts and sets it up to distribute the whole supply only in one period to a specific pool.
4. When that period starts, the attacker first uses 1 wei to vote for the selected pool, which will initialize the value of `accDebtPerShare` to a huge value.
5. When other legitimate users go to vote to that same pool, the transaction will revert due to overflow.

In step 4, the attacker votes using 1 wei to initialize `accDebtPerShare` to a huge value, which will happen here:

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/libraries/Rewarder2.sol#L161

```
    function updateAccDebtPerShare(Parameter storage rewarder, uint256
↪   totalSupply, uint256 totalRewards)
        internal
        returns (uint256)
    {
        uint256 debtPerShare = getDebtPerShare(totalSupply, totalRewards);

        if (block.timestamp > rewarder.lastUpdateTimestamp)
↪   rewarder.lastUpdateTimestamp = block.timestamp;

>>      return debtPerShare == 0 ? rewarder.accDebtPerShare :
↪   rewarder.accDebtPerShare += debtPerShare;
    }

    function getDebtPerShare(uint256 totalDeposit, uint256 totalRewards)
↪   internal pure returns (uint256) {
>>      return totalDeposit == 0 ? 0 : (totalRewards <<
↪   Constants.ACC_PRECISION_BITS) / totalDeposit;
    }
```

In the presented scenario, the value of `totalRewards` will be huge (close to `type(uint256).max`) and the value of `totalSupply` will only be 1, which will cause the `accDebtPerShare` ends up being a huge value.

Later, when legitimate voters try to vote for that same pool, the transaction will revert due to overflow because the operation of `deposit * accDebtPerShare` will result in a value higher than `type(uint256).max`:

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/libraries/Rewarder2.sol#L27-L29

```
function getDebt(uint256 accDebtPerShare, uint256 deposit) internal pure returns
↪  (uint256) {
    return (deposit * accDebtPerShare) >> Constants.ACC_PRECISION_BITS;
}
```

Impact

By executing this sequence, an attacker can block all votes to specific pools during specific periods. If wanted, an attacker could block ALL votes to ALL pools during ALL periods, and the bug can only be resolved by upgrading the contracts or deploying new ones.

There aren't any external requirements or conditions for this attack to be executed, which means that the voting functionality can be gamed or completely hijacked at any point in time.

PoC

The following PoC can be pasted in the file called `BribeRewarder.t.sol` and can be executed by running the command `forge test --mt testOverflowRewards`.

```
function testOverflowRewards() public {
    // Create custom token
    IERC20 customRewardToken = IERC20(new ERC20Mock("Custom Reward Token",
↪   "CRT", 18));

    // Create rewarder with custom token
    rewarder =
↪   BribeRewarder(payable(address(factory.createBribeRewarder(customRewardToken,
↪   pool))));

    // Mint the max amount of custom token to rewarder
    ERC20Mock(address(customRewardToken)).mint(address(rewarder),
↪   type(uint256).max);

    // Start bribes
    rewarder.bribe(1, 1, type(uint256).max);

    // Start period
    _voterMock.setCurrentPeriod(1);
    _voterMock.setStartAndEndTime(0, 2 weeks);

    vm.warp(block.timestamp + 10);
```

```
    // Vote with 1 wei to initialize `accDebtPerShare` to a huge value
    vm.prank(address(_voterMock));
    rewarder.deposit(1, 1, 1);

    vm.warp(block.timestamp + 1 days);

    // Try to vote with 1e18 -- It overflows
    vm.prank(address(_voterMock));
    vm.expectRevert(stdError.arithmeticError);
    rewarder.deposit(1, 1, 1e18);

    // Try to vote with 1e15 -- Still overflowing
    vm.prank(address(_voterMock));
    vm.expectRevert(stdError.arithmeticError);
    rewarder.deposit(1, 1, 1e15);

    // Try now for a bigger vote -- Still overflowing
    vm.prank(address(_voterMock));
    vm.expectRevert(stdError.arithmeticError);
    rewarder.deposit(1, 1, 1_000e18);
}
```

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/libraries/Rewarder2.sol#L28

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to create a whitelist of tokens that limits the tokens that can be used as rewards for bribing. This way, users won't be able to distribute a huge amount of tokens and block all the voting mechanisms.

Discussion

0xSmartContract

As a result of detailed analysis with LSW, the issue was determined as Medium.

The critical function is `_notifyBribes` in the `Voter` contract, which calls the `deposit` function in `BribeRewarder` contracts.

An attacker can create a custom token with a maximum supply and distribute it to a `BribeRewarder` contract, leading to an overflow when legitimate users try to vote.

- The PoC correctly sets up the scenario where an overflow error can occur. The critical part of the PoC is the initialization of `accDebtPerShare` with a large value.
- LSW mentioned that the PoC as provided might have issues, and suggest changing `uint256` to `uint192` for it to work correctly. This change makes sense as it ensures the `accDebtPerShare` calculation does not overflow.

1. `_notifyBribes()`

```
function _notifyBribes(uint256 periodId, address pool, uint256 tokenId,
↪  uint256 deltaAmount) private {
    IBribeRewarder[] storage rewarders = _bribesPerPriod[periodId][pool];
    for (uint256 i = 0; i < rewarders.length; ++i) {
        if (address(rewarders[i]) != address(0)) {
            rewarders[i].deposit(periodId, tokenId, deltaAmount);
            _userBribesPerPeriod[periodId][tokenId].push(rewarders[i]);
        }
    }
}
```

This function loops through `BribeRewarder` contracts and calls the `deposit` function.

2. `updateAccDebtPerShare()`

```
function updateAccDebtPerShare(Parameter storage rewarder, uint256 totalSupply,
↪  uint256 totalRewards)
    internal
    returns (uint256)
{
    uint256 debtPerShare = getDebtPerShare(totalSupply, totalRewards);

    if (block.timestamp > rewarder.lastUpdateTimestamp)
↪   rewarder.lastUpdateTimestamp = block.timestamp;

    return debtPerShare == 0 ? rewarder.accDebtPerShare :
↪   rewarder.accDebtPerShare += debtPerShare;
}

function getDebtPerShare(uint256 totalDeposit, uint256 totalRewards) internal
↪  pure returns (uint256) {
    return totalDeposit == 0 ? 0 : (totalRewards <<
↪   Constants.ACC_PRECISION_BITS) / totalDeposit;
}
```

- The `getDebtPerShare` function can produce a large value if `totalDeposit` is very small (e.g., 1 wei) and `totalRewards` is very large (`type(uint256).max`).
  - This large value can then cause overflow in subsequent calculations in `updateAccDebtPerShare`.

3. `getDebt()`

```
function getDebt(uint256 accDebtPerShare, uint256 deposit) internal pure
↪  returns (uint256) {
    return (deposit * accDebtPerShare) >> Constants.ACC_PRECISION_BITS;
}
```

This function performs the multiplication that can overflow if `accDebtPerShare` is too large.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/35

santipu03

Escalate

I believe this issue warrants high severity.

As I stated in the "impact" section of the report, this bug will allow an attacker to block all votes in some pools during all the periods, if wanted. We must consider that this impact isn't only a DoS, but an attack that will steal most rewards. That is because the voting of one period will directly translate into the allocated weights on the next period (the code for this is in js/sync-farms.js). An attacker can trigger this bug to block votes to specific pools, and that will result in the pools that benefit the attacker to earn most of the Masterchef rewards on the next period.

On the other hand, the only "limitation" that this vulnerability has is that the attacker must be able to add rewarders to the pools that are going to be DoSed from voting. And I don't believe that this limitation can be considered *extensive*, as the Sherlock docs describe.

Given the significant direct impact and minimal limitations associated with this issue, it should be classified as high severity.

sherlock-admin3

> Escalate
>
> I believe this issue warrants high severity.
>
> As I stated in the "impact" section of the report, this bug will allow an attacker to block all votes in some pools during all the periods, if wanted.

> We must consider that this impact isn't only a DoS, but an attack that will steal most rewards. That is because the voting of one period will directly translate into the allocated weights on the next period (the code for this is in js/sync-farms.js). An attacker can trigger this bug to block votes to specific pools, and that will result in the pools that benefit the attacker to earn most of the Masterchef rewards on the next period.
>
> On the other hand, the only "limitation" that this vulnerability has is that the attacker must be able to add rewarders to the pools that are going to be DoSed from voting. And I don't believe that this limitation can be considered *extensive*, as the Sherlock docs describe.
>
> Given the significant direct impact and minimal limitations associated with this issue, it should be classified as high severity.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

iamnmt

Escalate

I think this issue should be a duplicate of #190

I am arguing that both #190 and this issue are having the same root cause, which is permissionless bribe rewarder registration.

But this issue has a different impact from the impact of #190.

If the HOJ accepted that #190 and this issue are having the same root cause, then it would come down to if the impact of this issue is enough for it to be a separate issue basing on the duplication rule.

sherlock-admin3

> Escalate
>
> I think this issue should be a duplicate of #190
>
> I am arguing that both #190 and this issue are having the same root cause, which is permissionless bribe rewarder registration.
>
> But this issue has a different impact from the impact of #190.
>
> If the HOJ accepted that #190 and this issue are having the same root cause, then it would come down to if the impact of this issue is enough for it to be a separate issue basing on the duplication rule.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Oot2k

I completely agree with @iamnmt comment!

chinmay-farkya

This is rather a duplicate of #545. detailed comment here : https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/650#issuecomment-2263537234

santipu03

I disagree that this issue (237) and issue #190 have the same root cause.

I believe we can all agree that the root cause of issue #190 is the permissionless bribe rewarder registration.

However, the root cause of this issue is the missing checks for distributing huge amounts of tokens, which will later cause the overflow. To fix this issue we can simply implement a check on `BribeRewarder::_bribe` that ensures that the amount of tokens distributed per period doesn't surpass a limit, for example `type(uint128).max`.

Here is an example of the fix:

```
    function _bribe(uint256 startId, uint256 lastId, uint256 amountPerPeriod)
↪   internal {
        // ...

+       require(amountPerPeriod < type(uint128).max)

        // ...
    }
```

In conclusion, fixing issue #190 would fix this issue (because it's a broader fix that affects other issues) but fixing the root cause of this issue won't fix issue #190.

IIIIHunterIIII

want to add that Dup rules group issues to the higher severity ones,

> If the HOJ accepted that https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/190 and this issue are having the same root cause, then it would come down to if the impact of this issue is enough for it to be a separate issue basing on the duplication rule.

so the above statement is incorrect due to the fact that Dups rules has nothing to do with different severity as long as both reports are valid.

0xSmartContract

Duplicate #190 and #237 is a detail that we have discussed, analyzed in detail and know since the audit (we did not see it during Escalate). We discussed the issue in detail with LSW.

My final opinion is; this issue should be a duplicate of #190

Both findings indicate that the attacker can manipulate the system by using special or worthless tokens. These tokens can be used either by printing very large amounts or by adding them to BribeRewarder contracts while they are worthless.

Finding #190 indicates that the attacker can fill BribeRewarder contracts using worthless tokens, preventing other users from adding their real reward contracts.

Finding #237 indicates that the attacker can create an overflow error by using an excessively large amount of tokens, thus preventing voting for a specific pool.

Both findings carry the risk of a Denial of Service (DoS) attack. The attacker can disable the system by blocking a specific pool or the entire voting mechanism.Both findings indicate that the reward distribution mechanism can be manipulated by the attacker.

santipu03

The fact that issues #190 and #237 share some similarities, like the use of worthless tokens and the DoS risk, doesn't make them duplicates.

Each issue has a distinct underlying root cause, impact, and fix, so they should be treated separately.

WangSecurity

About the duplication escalation (@iamnmt). I believe the root cause of both issues are different and saying that it's permissionless adding of bribe rewarder is too broad:

1. This issue is about bribe rewarded having worthless tokens and the attacker minting `type(uint256).max` of that token supply, which leads to a revert here.
2. #190 root cause is that the attacker can create 5 bribe rewarders and link them to the same pool, not allowing to vote and create a new bribe rewarder to that pool, due to the revert here.

Hence, I believe these issues have different root causes and cannot be considered duplicates. @iamnmt escalation will be rejected.

About the severity escalation (@santipu03):

> As I stated in the "impact" section of the report, this bug will allow an attacker to block all votes in some pools during all the periods, if wanted. We must consider that this impact isn't only a DoS, but an attack that will steal most rewards. That is because the voting of one period will directly translate into the allocated weights on the next period (the code for this is in js/sync-farms.js). An attacker can trigger this bug to block votes to specific pools, and that will result in the pools that benefit the attacker to earn most of the Masterchef rewards on the next period.

Do I understand correctly that it's because the attacker is the only one who voted (their 1 wei vote) and no one else did?

Also, this issue doesn't lead to a loss of funds that were deposited by the users in the pool, i.e. Bob deposited 10 ETH, the attack happened, and Bob freely withdrew their 10 ETH, correct?

IIIIHunterIIII

If i can question a thing,

Why would you think its a broad statement if the solution (root cause) is the same, same place, same contract.

I don't know what was the solution, but if the solution was just turning bribes to be by admins(one solution solves both) doesn't this make it a dup?

Under the rule of same conceptual mistake

WangSecurity

> Why would you think its a broad statement if the solution (root cause) is the same, same place, same contract

Excuse me, I think I'm missing something, but the solution and the root cause are different things and I believe I explained in the previous comment why the root causes of the two issues are different. Is there something missing from my explanation?

> I don't know what was the solution, but if the solution was just turning bribes to be by admins(one solution solves both) doesn't this make it a dup?

As I understand it's the specific design of the protocol to allow everyone to create BribeRewarders and this is intended. Hence, I don't believe that making BribeRewarders is a "solution". Therefore, I don't see it as the same conceptual mistake because BribeRewarders are intended to be created by anyone. Hope that answers your questions.

@santipu03 waiting for your clarifications on the questions in the previous comment.

santipu03

@WangSecurity Hi sir, sorry for the delay.

> Do I understand correctly that it's because the attacker is the only one who voted (their 1 wei vote) and no one else did?

Kind of. The attacker will vote with 1 wei to some pools in order to block future votes to those pools. Then, the users will only have the option to vote for pools that the attacker doesn't block, and those pools will earn most of the rewards in the next epoch, stealing that yield.

For example, if we have only 2 pools (`pool1` and `pool2`) to vote, an attacker can exploit this bug to block all votes for `pool1`. Then, users will only have the option to vote for `pool2`, and that pool will steal most of the rewards in the next epoch, as determined by the script in js/sync-farms.js.

> Also, this issue doesn't lead to a loss of funds that were deposited by the users in the pool, i.e. Bob deposited 10 ETH, the attack happened, and Bob freely withdrew their 10 ETH, correct?

This issue won't affect the funds directly deposited in the pools, but it will allow an attacker to manipulate the voting system (the `Voter` contract) to steal most of the distributed yield in the next epoch. I consider that this issue causes a loss of funds because anyone can steal most of the rewards consistently until the bug is fixed.

All in all, I believe this issue deserves high severity given the potential losses and the low constraints.

I hope I've answered your questions, and I'll be glad to provide further clarification.

P.S. Revisiting the accepted issues, this report has a higher impact than issue #166, which is categorized as high severity. Just to ensure consistency in judging.

WangSecurity

Just to understand the issue more, what would happen if the attacker blocks both pools? Do they get all the rewards for that epoch?

santipu03

> Just to understand the issue more, what would happen if the attacker blocks both pools? Do they get all the rewards for that epoch?

I guess then both pools would receive the same amount of rewards because they both have 1 wei of votes each, which is the necessary amount of votes needed to block the following ones.

WangSecurity

Excuse me, I phrased the question poorly, I meant the attacker would receive all the rewards, correct (from both pools)?

santipu03

> Excuse me, I phrased the question poorly, I meant the attacker would receive all the rewards, correct (from both pools)?

That would depend on whether the attacker is the only staker for those pools within the `MasterChef` contract. In this scenario, both pools would receive the same amount of rewards, and the rewards would go to the users staking LP tokens for those pools.

The best impact that an attacker can achieve is to block the votes for all pools except for the one that benefits him the most, therefore earning most of the rewards within the `MasterChef` contract on the next epoch, stealing that yield from other users.

Does that answer your question? @WangSecurity

WangSecurity

I agree it's high severity. Indeed the attacker can cause a loss of funds to other users (in terms of rewards), while it doesn't have any extensive limitations.

Planning to reject the @iamnmt escalation, since it asks to duplicate this issue with another one, but accept @santipu03 escalation to upgrade the severity to high.

WangSecurity

Result: High Unique

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- santipu03: accepted
- iamnmt: rejected

WangSecurity

Based on this and this this will be the new main report for #545 issue family.

santipu03

Respectfully sir, I don't understand how it can be considered that issue #545 and this one have the same root cause.

Issue #545 root cause is the use of "weird ERC20" tokens within the protocol, as determined here:

> Here, the conceptual mistake is Weird tokens. Hence, planning to reject the escalation and leave the issue as it is.

However, the root cause of this issue is the missing checks for distributing huge amounts of tokens on the Rewarders. Unlike the issues related to #545, this issue does not involve any "weird ERC20" tokens.

Moreover, this issue can be fixed by simply adding an extra check in the `_bribe` function, as expressed here. The recommendation made in the original report wasn't quite accurate because I later realized that the protocol can easily fix this issue by adding this extra line of code instead of removing a core mechanic of the protocol, which is the permissionless use of tokens.

From my understanding of reading the escalation on issue #545, the decision to group all the family under the same conceptual mistake of "weird tokens" was made for the sake of simplicity and because all dups didn't really add more value to the protocol. However, I believe that broadening the conceptual mistake so that more issues fit into that category isn't the right decision given that this issue is pointing at a different problem than the family of #545.

The fact that applying a broad fix to issue #545 (limiting the use of tokens) would also fix this issue isn't enough to consider them duplicates IMO. It happened a ton of times that the developers of the protocol can fix different issues by applying a broad fix (e.g. removing a core mechanic of the protocol) but that doesn't make those issues duplicates because they have different root causes even if the fix ends up being the same.

@WangSecurity I kindly ask you to reconsider the decision.

chinmay-farkya

> However, the root cause of this issue is the missing checks for distributing huge amounts of tokens on the Rewarders.

I don't think this is correct. The root cause is "random tokens can be permissionlessly added to the system" in order to cause myriad of impacts listed in different issues under #545.

> Moreover, this issue can be fixed by simply adding an extra check in the _bribe function, as expressed https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/237#issuecomment-2263555440.

Every issue under #545 can have a range of fixes but that does not make this #237 different from the family as the root cause remains the same.

> the decision to group all the family under the same conceptual mistake of "weird tokens" was made for the sake of simplicity and because all dups didn't really add more value to the protocol

I don't think that this is true. The listings inside #545 family point to many different issues at different places. Examples are :

1. #231 pointing to tokens with large decimals, affecting MLUMStaking contract.

2. #296 showing how "large amount of a token getting deposited" causing issues, affecting Masterchef contract, which can also cause overflow : ie. the same impact as #237
3. #500 shows "tokens perceived as standard : like USDC" causing issues due to extraRewarder contract
4. #655 shows ctokens edge case issue when trying to transfer large amount of tokens : affecting multiple contracts
5. #499 shows tokens with low decimals create problems in MLUMStaking
6. #14 shows tokens with low decimals can create problems in calculations

Note that the high decimal tokens (one impact being an overflow : see point 2.) is also mentioned in many other duplications under #545 . So this issue #237 is not really different in pointing out that "a token with high circulating supply and that too controlled completely by the attacker" can be used for overflow DOS"z

> I believe that broadening the conceptual mistake so that more issues fit into that category isn't the right decision given that this issue is pointing at a different problem than the family of https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545.

We are not broadening it. it already a single conceptual mistake. You can see the reasoning under #545 by lead judge of grouping different issue categories into one becauseopf having the same root cause and same fix. Your issue #237 is not the only "type" there are many types already grouped into one because they all point to the use of custom tokens : which could have a hundred different problems that people have not listed in duplicates of #545 even.

> It happened a ton of times that the developers of the protocol can fix different issues by applying a broad fix (e.g. removing a core mechanic of the protocol) but that doesn't make those issues duplicates because they have different root causes even if the fix ends up being the same.

The case of custom tokens is different from what you are pointing to. Here the "allowing the use of custom tokens" is the root cause, which can have many different impacts.

santipu03

> I don't think this is correct. The root cause is "random tokens can be permissionlessly added to the system" in order to cause myriad of impacts listed in different issues under https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545.

To better understand the term "root cause," let's refer to the definition provided in the Sherlock documentation:

> [!IMPORTANT] Root Cause: The primary factor or a fundamental reason that imposes an unwanted outcome

The root cause of this issue cannot be "random tokens can be permissionlessly added to the system" because that isn't the primary factor that imposes an unwanted outcome. That is because the protocol can continue having permissionless tokens on the Rewarders and still fix this issue, and that's why that's not the primary factor.

The primary factor in this issue is the missing checks for distributing huge amounts of tokens on the Rewarders, which can be fixed by simply adding an extra line in the `_bribe` function. If that fix is applied, the issue is fixed because the primary factor has been fixed.

> We are not broadening it. it already a single conceptual mistake. You can see the reasoning under https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545 by lead judge of grouping different issue categories into one becauseopf having the same root cause and same fix. Your issue https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/237 is not the only "type" there are many types already grouped into one because they all point to the use of custom tokens : which could have a hundred different problems that people have not listed in duplicates of https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545 even.

That's incorrect. If you take a look at the escalation on issue #545, the conceptual mistake defined by the Lead Judge and by the HOJ has been the use of "weird tokens", you can check it here and here. Later, in your comment here you try to broaden the conceptual mistake to "allowing custom tokens to be linked to the briberewarder", which is different from the original one defined by the judges.

chinmay-farkya

> That is because the protocol can continue having permissionless tokens on the Rewarders and still fix this issue, and that's why that's not the primary factor.

Yes, but this statement is also true for many types I listed above in my comment, and others in the listed duplicates of #545. Many of those can be solved by other means while still having permissionlessness. That argument is not relevant to the discussion of "root cause" : the root cause is indeed the use of random tokens.

> The primary factor in this issue is the missing checks for distributing huge amounts of tokens on the Rewarders

I don't think this is true. "distributing huge amounts of tokens" does not amount to any relevant issue, it is only an issue when an attacker can embed a token whose circulating supply is so large upto type(uint256).max and is "fully controlled by him"

: which points back to allowing custom tokens into the system.

I think root cause and fix are the same as #545.

> Later, in your comment https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/650#issuecomment-2290595068 you try to broaden the conceptual mistake to "allowing custom tokens to be linked to the briberewarder"

That is exactly what a conceptual mistake of "allowing random tokens to be used" means !

santipu03

> Yes, but this statement is also true for many types I listed above in my comment, and others in the listed duplicates of https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545. Many of those can be solved by other means while still having permissionlessness.

I agree with this, and that's why those issues also have different root causes. In this specific case, the Lead Judge and Wang have decided to make an exception to group all of them under the same concept of "weird tokens", which I agree with for the sake of simplicity and fairness.

However, I don't believe broadening the "conceptual mistake" to fit even more issues (like this one) is fair. Because then, what is the limit of a "conceptual mistake"? Could we consider a "conceptual mistake" the use of rewarders at all within this protocol? Then, all issues related to the rewarders would also have to be grouped...

As you can see, the use of a "conceptual mistake" is pretty subjective and can lead to grouping most of the issues found in this contest. I believe the original decision made by Wang is fair, which only limits the conceptual mistake to the use of "weird tokens", leaving this issue out of that family given that it points to a different problem.

WangSecurity

After reading the discussion and additionally thinking about this issue, I agree my previous decision was fair, i.e. to keep it a separate issue.

Firstly, it's the protocol's design to allow exactly any token to be used for bribe rewards. Hence, saying that both this and #545 family can be fixed by adding a whitelist would be incorrect, because it's protocol intention to not have it. That's also the reason why I believe grouping this report with #545 under the root cause of "allowing to use any tokens for bribe rewards" is not correct, because the protocol wants to have any tokens as bribe rewards. Hence, I believe it would be fair to find a deeper root cause than the entire intended design.

Then, comes the question, why not separate all the issues under #545 into different families? Because all the reports under #545 address the issues related to the same class of tokens, i.e. tokens with weird traits. This report shows how this attack can be executed with a standard token.

Yes, this issue is about maliciously created tokens specifically to attack the protocol. But, it's still a standard one and doesn't have any weird traits from here.

Hence, I'm reverting to my initial decision that this should remain separate from #545.

Slavchew

> Then, comes the question, why not separate all the issues under https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545 into different families? Because all the reports under https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545 address the issues related to the same class of tokens, i.e. tokens with weird traits. This report shows how this attack can be executed with a standard token.
>
> Yes, this issue is about maliciously created tokens specifically to attack the protocol. But, it's still a standard one and doesn't have any weird traits from here.

Ok, if you consider all issues that mark `weird` traits from this repo. Why then reports that also show different issues and they are "standard" in your words not in this repo - https://github.com/d-xo/weird-erc20, are not separated issues?

All duplicates of #545 talk about - rebase, FOT, low/high decimals, blacklist, cUSDCv3, revert on transfer/approve, and the rest of the repo.

But there is one different issue that is not marked in this repo and from your clarification above it should also be a separate issue if we are going to stick to the same judging rules and not favor any Watson or report.

This is issue #499, and all its dups - #93, #148, #199.

Here is even an example of a codebase utilizing a fix for this type of tokens - https://github.com/Cyfrin/2024-07-zaros/blob/d687fe96bb7ace8652778797052a38763fbcbb1b/src/perpetuals/branches/GlobalConfigurationBranch.sol#L253

So I will kindly ask you to check again. If you going to separate this from #545 issue by this argument - https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/237#issuecomment-2295331853, then #499 is definitely a separate one as well.

iamnmt

@WangSecurity

> Firstly, it's the protocol's design to allow exactly any token to be used for bribe rewards. Hence, saying that both this and https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545 family can be fixed by adding a whitelist would be incorrect, because it's protocol intention to not have it.

I think arguing that design decisions is not a bug is an invalid argument. Basing on this rule:

> 9. **Design decisions** are not valid issues. Even if the design is suboptimal, but doesn't imply any loss of funds, these issues are considered informational.

This rule can be interpret that if design decisions cause loss of funds then it is a valid issue. So, I believe the root cause for all the mentioned issue is the design decision to allow "any type of ERC20 token" (quoting from the contest's `README`).

Let's say the #237 issue is "about maliciously created tokens specifically to attack the protocol". Consider the following issues about a custom "standard" ERC20 token:

- An ERC20 token that will revert on `transferFrom` if the `from` address is not the owner, the owner still can create a bribe rewarder but no one can claim the bribe rewards.
- An ERC20 token that will set `_balances[to]` to `type(uint256).max` on `transferFrom`. An ERC20 token will always return balance of the bribe rewarder as `type(uint256).max`. These issues are the same as #237.

Should these issues be considered valid? Do these issues add any value to the protocol?

Separating issues "about maliciously created tokens specifically to attack the protocol" from #545 will defeat the purpose of grouping issues in #545 in the first place, which is

> **Unified Theme and Clarity:** By categorizing these issues together, the report maintains a coherent theme, making it easier for auditers and Sponsor to understand the underlying problem of dealing with unconventional ERC20 tokens. **Specification Looseness:** The ERC20 specification is known to be loosely defined, leading to diverse implementations. Many tokens deviate from standard behaviors, causing unexpected issues when integrated into smart contracts.

Also, note that custom "standard" ERC20 tokens is a superset of ERC20 tokens with weird traits. Should the issues related to ERC20 tokens with weird traits be grouped into the issues related to custom "standard" ERC20 tokens?

chinmay-farkya

> which only limits the conceptual mistake to the use of "weird tokens", leaving this issue out of that family given that it points to a different problem.

If this points to a different problem, then many issues under #545 point to a different problem as well : stated in my comment https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/237#issuecomment-2295192020

Now for your comment @WangSecurity

> Because all the reports under https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545 address the issues related to the same class of tokens, i.e. tokens with weird traits. This report shows how this attack can be executed with a standard token.

This issue's mentioned attack can not be executed with a standard token ! A token which is controlled by the attacker is not a standard token at all !

Think about this : the attacker needs to control all of the circulating supply of the token and push it as rewards inside the briberewarder : how is this defined as a standard token ?

> Hence, I believe it would be fair to find a deeper root cause than the entire intended design.

If we do intend to find a deeper root cause, then the next level of root cause is not "distribution of large amount of rewards" but it is " permissionless registration of a briberewarder" from #190 since this issue stems from the fact that the briberewarder added by attackers can cause DOS in a variety of ways, with different ways to attack but the same root cause of allowing anyone to register a briberewarder in the voter contract, the exact fix of not allowing random people to add briberewarders and the same impact of DOS! In that case it should be a duplicate of #190

> Because all the reports under https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545 address the issues related to the same class of tokens, i.e. tokens with weird traits

Again, what is described as weird trait and what is not is also subjective. We can't be leaving out a single issue (which indeed belongs to the class of custom tokens : an attacker may tomorrow come up with a custom token with random functionality, and just because it is not listed in the weird erc20 repo, it will be deemed as a standard token ? Sounds illogical.) , while counting everything else under the sun as having a root cause of "permissionless use of custom tokens"

> I believe grouping this report with https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545 under the root cause of "allowing to use any tokens for bribe rewards" is not correct, because the protocol wants to have any tokens as bribe rewards

BribeRewarder also does have other issues of custom tokens : as mentioned in my comment https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/237#issuecomment-2295192020. So to say that this report is different from that type is incorrect.

I 100 % agree with @Slavchew and @iamnmt 's examples above. All these issues have the same cause and the same exact fix. (even santipu's report proposes the same fix of not allowing permissionless tokens).

This one issue is also stemming from the same thing, why are we leaving this one outside the theme of conceptual mistake ?

> Yes, this issue is about maliciously created tokens specifically to attack the protocol. But, it's still a standard one and doesn't have any weird traits from here.

I actually found something from the repo. This statement is incorrect.

I will point you to this :
https://github.com/d-xo/weird-erc20?tab=readme-ov-file#flash-mintable-tokens

> Flash Mintable Tokens Some tokens (e.g. DAI) allow for so called "flash minting", which allows tokens to be minted for the duration of one transaction only, provided they are returned to the token contract by the end of the transaction. This is similar to a flash loan, but does not require the tokens that are to be lent to exist before the start of the transaction. A token that can be flash minted could potentially have a total supply of max uint256.

If you re-analyze the original submission from santipu, you will find striking similarities between the custom token the report mentions and the category of flash minted tokens.

1. This section mentions that some tokens can have totalsupply as uint(256).max : the report is also showing an attack with a custom token with circulating totalsupply of uint256.max
2. This section mentions that a token could be minted in large quantities by a single user : the report is also showing the same thing ie. large amounts minted by a single user

So I think this does fall into this category of flash mintable weird tokens ! So this is not any different from the category of "tokens with weird traits"

santipu03

@Slavchew

I see your point about issue #499, which describes an issue about a token that doesn't implement the `decimals` function. However, I agree with the Lead Judge in

this comment because that kind of token isn't part of the default ERC20 implementation.

On the other hand, my report describes an issue with a completely normal token, meaning it can use the OpenZeppelin ERC20 implementation.

@iamnmt

All the examples that you've given as custom "standard" tokens share the same similarities, which is that they still have "weird" traits and that's why they should be grouped under the "weird tokens" family. In my report, there's no "weird trait" involved because the attacker is using the default ERC20 implementation from OZ.

@chinmay-farkya

I believe you're misunderstanding the concept of "standard" tokens here. From what I've seen in past judgements on Sherlock, a "standard" token isn't just a token that follows the ERC20 standard, but is one that uses the default ERC20 implementation (e.g. OZ).

Also, regarding the "flash mintable" tokens. As the definition states, a "flash mintable" token can have a large number of tokens minted only for the duration of one transaction, and those tokens have to be returned at the end of the transaction. As you can see from simply reading the definition, the token I've used for this report IS NOT flash minted, but a normal token using the default ERC20 implementation.

Slavchew

It was never mentioned what a standard token is, I just point the facts that #499 should also be valid based on @WangSecurity idea of grouping all "weird" tokens based on the repo he provided.

If there will be a separate issue for, so #499 should also be. Same judging rules should apply for all parties.

Also about your answer to @chinmay-farkya, do not provide historical ideas of how was been before. Every contest has its different judging approach from what I've seen and if @WangSecurity said that weird is from the repo, then only issues specifying that type should be grouped together.

So @WangSecurity should decide if going to differentiate weird from standard based on the repo then #499 should also be valid, if not then he is contradicting himself and flavor this issue most than others.

santipu03

@Slavchew

From the Sherlock docs:

> Non-Standard tokens: Issues related to tokens with non-standard behaviors, such as weird-tokens are not considered valid by default unless these tokens are explicitly mentioned in the README.

As the rules say, a non-standard token can be defined in the weird-ERC20 repo but it's not limited by that. This means that a token with a non-standard behavior, such as not implementing the `decimals` function, can still be considered a "weird token" even if it's not actually in the weird-ERC20 repo.

My reply to Chinmay about the definition of a "standard token" it's not only based on past judgments but is also explicitly defined in the Sherlock rules.

Slavchew

Not implementing decimals is not `weird token`, first as you pointed the docs, strictly limited the repo from where weird tokens are accepted, second this part of the docs is not overwritten in the README of this contest, so only the docs stay as last valid rules and lastly, if you read #499, will find out the decimals is not part of the EIP and its optional - https://eips.ethereum.org/EIPS/eip-20#decimals.

I'm waiting for @WangSecurity to answer the previous comments first since we only speculating on what should and shouldn't `standard` and `weird token` be.

santipu03

> Not implementing decimals is not weird token

I believe you're mistaken here, take a look at the README from the weird-erc20:

> This repository contains minimal example implementations in Solidity of ERC20 tokens with behaviour that may be surprising or unexpected.

An ERC20 implementation that does not implement the `decimals` function can be categorized as a "weird behavior" given that it's surprising and unexpected. This "weird behavior" is not in the weird-erc20 repo probably because a token like this has never been created in the blockchain.

On the other hand, as I've repeatedly stated, my issue doesn't need any token with "weird behavior" at all, given that it uses the default ERC20 implementation from OZ.

Slavchew

man, you're just making some assumption about what the writer said and how to interpret it, but it's very relative, as far as I understand from what @WangSecurity wrote, we are looking at what's mentioned there, there's only about low/high decimals.

He should make up his mind as HOJ, how this must be interpreted, not just us writing pointless comments about what should be interpreted how, and why.

WangSecurity

> Ok, if you consider all issues that mark weird traits from this repo. Why then reports that also show different issues and they are "standard" in your words not in this repo - https://github.com/d-xo/weird-erc20, are not separated issues?

I've already answered this question under the #545 escalation, I believe it's fair to group issues about tokens with weird traits in one conceptual mistake, while this report is about a completely standard token without any weird traits.

> I think arguing that design decisions is not a bug is an invalid argument

I didn't say it's "not a bug".

> This rule can be interpret that if design decisions cause loss of funds then it is a valid issue. So, I believe the root cause for all the mentioned issue is the design decision to allow "any type of ERC20 token" (quoting from the contest's README).

Basically, this has already been discussed above and I've explained why I decided to group issues in this way.

> Should these issues be considered valid? Do these issues add any value to the protocol?

I don't think we talk about the validity of those issues here. To clarify, both this and #545 families are valid, excuse me if there was confusion.

> Separating issues "about maliciously created tokens specifically to attack the protocol" from https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545 will defeat the purpose of grouping issues in https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545 in the first place, which is Also, note that custom "standard" ERC20 tokens is a superset of ERC20 tokens with weird traits. Should the issues related to ERC20 tokens with weird traits be grouped into the issues related to custom "standard" ERC20 tokens

My view is that #545 is about tokens with weird traits from the this, while the token "whose in full control of one user" or "the entire supply is minted" is not there.

> This issue's mentioned attack can not be executed with a standard token ! A token which is controlled by the attacker is not a standard token at all ! Think about this : the attacker needs to control all of the circulating supply of the token and push it as rewards inside the briberewarder : how is this defined as a standard token ?

That's a very fair argument, to be honest. But, the weird traits are only the ones mentioned in the repo linked above, e.g. low decimals, high decimals, blacklist mechanism, revert on transfer failure, etc. This attack is possible with a token

without any of these traits. Yes, the token is created maliciously only to execute an attack on MagicSea, still it doesn't require any of the weird traits.

> If we do intend to find a deeper root cause, then the next level of root cause is not "distribution of large amount of rewards" but it is " permissionless registration of a briberewarder" from https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/190 since this issue stems from the fact that the briberewarder added by attackers can cause DOS in a variety of ways, with different ways to attack but the same root cause of allowing anyone to register a briberewarder in the voter contract, the exact fix of not allowing random people to add briberewarders and the same impact of DOS! In that case it should be a duplicate of https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/190

We've already discussed this issue being a duplicate of #190, which I disagree with that's why that escalation was rejected.

> Again, what is described as weird trait and what is not is also subjective. We can't be leaving out a single issue (which indeed belongs to the class of custom tokens : an attacker may tomorrow come up with a custom token with random functionality, and just because it is not listed in the weird erc20 repo, it will be deemed as a standard token ? Sounds illogical.) , while counting everything else under the sun as having a root cause of "permissionless use of custom tokens"

I think we look at the weird and standard tokens differently here. As I've said before, weird tokens are only the ones with weird traits from the d-xo repo linked above. Standard tokens are the ones without these traits. Custom token is unfortunately not in the weird tokens repo. Hence, it would be me going over the rules to add another weird trait mid-escalation. This attack is possible with a token without any of the weird traits, while the token is created maliciously to exploit the protocol.

> If you re-analyze the original submission from santipu, you will find striking similarities between the custom token the report mentions and the category of flash minted tokens

Interesting comparison. It indeed looks a bit similar to flash mint mechanic, but I still believe this issue is still possible with a standard token (i.e. a token without weird traits, "custom token" is not a weird trait).

> It was never mentioned what a standard token is,

It has always been known what the standard token is based on the rule quoted by @santipu03 in the previous comment. This repo from d-xo is specifically linked in our docs as a reference which traits are considered weird.

About 0-decimal tokens. I believe 0-decimal tokens are equal to low decimals and

is a weird trait. The standard implementation is considered 18 decimals, if <18 decimals then it's low decimals, if >18 decimals then it's high decimals.

Hope that answers all the questions from the previous comments. If I missed something and any of your concerns are unanswered, let me know if you need additional clarification. The decision remains, leave this as a separate issue since it involves a token without any of the weird traits mentioned in the weird tokens repo specifically linked in our rules.

Slavchew

Since you confirm that only weird traits from the mentioned repo will should be grouped together, then how token without decimals is low decimals. That is not true. The repo explicitly stating for tokens with lower decimals which will cause calculation issues not for token that do not have implemented decimals() and will revert when the function is called.

Please review this again.

WangSecurity

I believe any number of decimals except 18 is a weird trait. If it's >18, then it's high decimals. If it's <18 then it's low decimals. From my perspective is quite low relatively to 18. That's why I treat it as a weird token. The decision remains the same, keep it a separate issue.

chinmay-farkya

Hey @WangSecurity I will subtract all other arguments and try to draw your attention to this one thing :

In your comment here : https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/237#issuecomment-2295331853 you state that

> Yes, this issue is about maliciously created tokens specifically to attack the protocol. But, it's still a standard one and doesn't have any weird traits from here.

But as I have pointed out in my comment here : https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/237#issuecomment-2295746977

The token used for the attack in this report #237 has two "weird traits" that have been mentioned in the weird erc20 repo :

1. Token with a totalsupply of uint256.max
2. token which can be minted by an attacker in large quantities easily

The definition of a flash mintable token does not matter here, as we are only talking about "weird traits" which can take on various permutations and combinations of the examples mentioned.

For example, Low decimals section : https://github.com/d-xo/weird-erc20?tab=readme-ov-file#low-decimals mentions two examples of tokens including 6 decimals and 2, but it is common sense that any other number < 18 is also a weird token.

Similarly, a token with the traits I mentioned above is a weird token because it has the weird traits, regardless of if it is termed as flash mintable or not. It can be any variation of the same traits category.

Now to answer your comment above =>

> That's a very fair argument, to be honest. But, the weird traits are only the ones mentioned in the repo linked above, e.g. low decimals, high decimals, blacklist mechanism, revert on transfer failure, etc. This attack is possible with a token without any of these traits. Yes, the token is created maliciously only to execute an attack on MagicSea, still it doesn't require any of the weird traits.

That is not true because it needs to have these two weird traits for the attack to succeed and they have also been mentioned in the repo.

> Interesting comparison. It indeed looks a bit similar to flash mint mechanic, but I still believe this issue is still possible with a standard token (i.e. a token without weird traits, "custom token" is not a weird trait).

Okay, I think that is unfair, and still totalsupply at uint256.max and large quantity of minting allowed to a person are "weird traits" mentioned in the repo.

santipu03

@chinmay-farkya

A token that can have a supply of `uint256.max` isn't a "weird" token because all implementations of ERC20 have this characteristic. On the other hand, a token that allows flash minting is indeed a "weird" token because it's a surprising and unexpected behavior, just as the weird-erc20 repo defines.

WangSecurity

> The token used for the attack in this report https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/237 has two "weird traits" that have been mentioned in the weird erc20 repo : Token with a totalsupply of uint256.max token which can be minted by an attacker in large quantities easily The definition of a flash mintable token does not matter here, as we are only talking about "weird traits" which can take on various permutations and combinations of the examples mentioned

I assume you're talking about flash mintable tokens. Flash-mintable tokens `allow tokens to be minted for the duration of one transaction only, provided they are returned to the token contract by the end of the transaction.` I don't see

how it's involved here in this issue. Yes, flash-mintable tokens can be minted in large quantities with a potential total supply for one transaction. The two traits you've described are not separate weird traits, but the parts of the weird traits of flash-mintable tokens. I believe the definition of the flash-mintable token matters here since the flash-mintable token is a weird trait.

Moreover, reaching the total supply of max uint256 value can be done with any token. Yes, it's not very realistic, but it's possible on many tokens without any weird traits. The same with minting large quantities, you can mint a large quantity on many of the tokens. That's why I disagree that having these 2 traits makes the token weird.

> That is not true because it needs to have these two weird traits for the attack to succeed and they have also been mentioned in the repo Okay, I think that is unfair, and still totalsupply at uint256.max and large quantity of minting allowed to a person are "weird traits" mentioned in the repo

Hope my words above explain why these are not weird traits. The decision remains the same, keep this a separate issue.

chinmay-farkya

> I assume you're talking about flash mintable tokens. Flash-mintable tokens allow tokens to be minted for the duration of one transaction only, provided they are returned to the token contract by the end of the transaction. I don't see how it's involved here in this issue.

No, I'm not specifically talking about flash-mintable tokens, I'm just trying to say that these are weird traits in general.

> The two traits you've described are not separate weird traits, but the parts of the weird traits of flash-mintable tokens

IMO that is not correct, as the traits can be present in any random tokens. What if a token comes up that has large totalSupply and also has low decimals == 2 , then because the weird erc20 repo does not mention such a category that is "LOW DECIMAL Flash-mintable tokens" : does that make it a standard token ?

The point I'm making here is that it still remains a weird token because the traits could be selectively present/absent in various permutations and combinations of these features in the wild, but still that doesn't make it a standard token.

> Moreover, reaching the total supply of max uint256 value can be done with any token. Yes, it's not very realistic, but it's possible on many tokens without any weird traits. The same with minting large quantities, you can mint a large quantity on many of the tokens. That's why I disagree that having these 2 traits makes the token weird.

"Yes its not very realistic" : that is the whole point of calling these traits weird and

the tokens as non-standard !!

Practically speaking, the token required in this issue report needs to have those weird traits that you refuse to acknowledge as weird traits, even though they are mentioned in the repo. I'd like to see examples of actual practical tokens in the market that have a totalsupply of uint256.max : I dont think any token has such a large circulating supply, and that's what the definition of a weird token in my mind is : unrealistic properties.

I sum up my arguments over here. I really think its unfair to argue over semantics in a custom malicious token vs "weird tokens" : they are the same because weird tokens are just a subset of the possibilities custom tokens have and the repo can never be a full representation of it.

the rest is your call now.

WangSecurity

> No, I'm not specifically talking about flash-mintable tokens, I'm just trying to say that these are weird traits in general.

But you've got these 2 "weird traits" from the flash-mintable tokens definition. And I disagree these are "weird traits in general". You can mint a large quantity of any token. Total supply reaching uint256.max is not weird as well as it can happen with many tokens. The fact that there wasn't such a case, it doesn't mean no standard tokens will not reach this state. These 2 "weird traits" are only a definition of flash-mintable tokens, i.e. you can mint a large quantity of tokens (up to uint256.max) for only one transaction. These are not weird traits on their own, they are only a definition of the flash-mintable tokens.

> IMO that is not correct, as the traits can be present in any random tokens. What if a token comes up that has large totalSupply and also has low decimals == 2 , then because the weird erc20 repo does not mention such a category that is "LOW DECIMAL Flash-mintable tokens" : does that make it a standard token ?

No, this is the token with 2 different weird traits. I don't understand how this question is relevant to discussion, these are two distinct weird traits, while the other 2 are just normal characteristics of the token.

> The point I'm making here is that it still remains a weird token because the traits could be selectively present/absent in various permutations and combinations of these features in the wild, but still that doesn't make it a standard token.

I agree, and I didn't say "it's not a weird token because the combination of these 2 traits are not mentioned to be weird". Excuse me if it seems so. What I'm saying is that these 2 are not weird traits in the first place.

> Practically speaking, the token required in this issue report needs to have those weird traits that you refuse to acknowledge as weird traits, even though they are mentioned in the repo. I'd like to see examples of actual practical tokens in the market that have a totalsupply of uint256.max : I dont think any token has such a large circulating supply, and that's what the definition of a weird token in my mind is : unrealistic properties.

I've already touched it above. Yes, there no such tokens with this large total supply now, but how reaching this total supply makes a token weird? Ethereum doesn't have a fixed total supply, do you think if Ethereum reaches the total supply of uint256.max then it's weird? If stETH reaches the total supply of uint256.max, would it be a weird token? Same for WETH? PEPE? Or any other token with no other weird trait essentially becomes weird if it reaches the total supply of uint256.max? Unfortunately, I look at it differently and I don't think that token being able to reach such total supply is a weird trait.

This is my final call, I believe these are not weird traits, the token used in this issue is not a weird token. Hence, I believe it's a separate issue and I'm planning to finalise the contest today since all the other escalations are resolved.

IIIIHunterIIII

With all respect, @WangSecurity

> do you think if Ethereum reaches the total supply of uint256.max then it's weird? If stETH reaches the total supply of uint256.max,

i know one thing for sure, that any thing in the world other than atoms reaching uint256.max is weird to me

The world of defi will collapse if any token you mentioned reaches uint256.max and all of auditors here know this,

- I really want to have a stable ground here, you really see uint256.max is not weird?

WangSecurity

Excuse me, indeed my above comment was a bit off. It's of course impossible to know what happens if any of these major tokens reach this value. What I meant is that all of them can reach it, i.e. there is no cap (as far as I know) on their total supply and it's a normal characteristic for a token. That's what I meant and excuse me for a very bad phrasing and example.

WangSecurity

Planning to finalise the contest tomorrow 12 pm CEST to not withhold the contest results since the other escalations are resolved.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-1: New staking positions still gets the full reward amount as with old stakings, diluting rewards for old stakers

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/74

Found by

ChinmayF, DPS, LeFy, PUSH0, Yashar, dany.armstrong90, minhquanym, oualidpro, qmdddd, scammed, sh0velware, utsav, zarkk01

Summary

New staking positions still gets the full reward amount as with old stakings, diluting rewards for old stakers. Furthermore, due to the way multipliers are calculated, extremely short stakings are still very effective in stealing long-term stakers' rewards.

Vulnerability Detail

In the Magic LUM Staking system, users can lock-stake MLUM in exchange for voting power, as well as a share of the protocol revenue. As per the docs:

> You can stake and lock your Magic LUM token in the Magic LUM staking pools to benefit from protocol profits. All protocol returns like a part of the trading fees, Fairlaunch listing fees, and NFT trading fees flow into the staking pool in form of USDC.
>
> Rewards are distributed every few days, and you can Claim at any time.

However, when rewards are distributed, new and old staking positions are treated alike, and immediately receive the same rewards as it is distributed.

Thus, anyone can stake MLUM for a short duration as rewards are distributed, and siphon away rewards from long-term stakers. Staking for a few days still gets almost the same multiplier as staking for a year, and the profitability can be calculated and timed by calculating the protocol's revenue using various offchain methods (e.g. watching the total trade volume in each time intervals).

Consider the following scenario:

- Alice stakes 50 MLUM for a year.
- Bob has 50 MLUM but hasn't staked.
- Bob notices that there is a spike in trading activity, and the protocol is gaining a lot of trading volume in a short time (thereby gaining a lot of revenue).

- Bob stakes 50 MLUM for 1 second.
- As soon as the rewards are distributed, Bob can harvest his part immediately.

Note that expired positions, while should not be able to vote, still accrue rewards. Thus Bob can just leave the position there and withdraw whenever he wants to without watching the admin actions. A more sophisticated attack involves front-running the admin reward distribution to siphon rewards, then unstake right away.

PoC

Due to the way multipliers are calculated, 1-year lock positions are only at most 3 times stronger than a 1-second lock for the same amount of MLUM.

The following coded PoC shows the given scenario, where Bob is able to siphon 25% of the rewards away by staking for a duration of a single second and leave it there.

Paste the following test into `MlumStaking.t.sol`, and run it by `forge test --match-test testPoCHarvestDilute -vv`:

```
function testPoCHarvestDilute() public {
    _stakingToken.mint(ALICE, 100 ether);
    _stakingToken.mint(BOB, 100 ether);

    vm.startPrank(ALICE);
    _stakingToken.approve(address(_pool), 50 ether);
    _pool.createPosition(50 ether, 365 days);
    vm.stopPrank();

    vm.startPrank(BOB);
    _stakingToken.approve(address(_pool), 50 ether);
    _pool.createPosition(50 ether, 1);
    vm.stopPrank();

    _rewardToken.mint(address(_pool), 100 ether);

    skip(100); // Bob can stake anytime and then just wait

    vm.prank(ALICE);
    _pool.harvestPosition(1);
    vm.prank(BOB);
    _pool.harvestPosition(2);

    console.logUint(_rewardToken.balanceOf(ALICE));
    console.logUint(_rewardToken.balanceOf(BOB));
```

```
}
```

The test logs will be

i.e. Bob was able to siphon 25% of the rewards.

Impact

Staking rewards can be stolen from honest stakers.

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/MlumStaking.sol#L354

Tool used

Manual Review

Recommendation

When new positions are created, their position should be recorded, but their amount with multipliers should be summed up and queued until the next reward distribution.

When the admin distributes rewards, there should be a function that first updates the pool, then add the queued amounts into staking. That way, newly created positions can still vote, but they do not accrue rewards for the immediate following distribution (only the next one onwards).

- One must note down the timestamp that the position was created (as well as the timestamp the rewards were last distributed), so that when the position unstakes, the contract knows whether to burn the unstaked shares from the queued shares pool or the active shares pool.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/7

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-2: `MlumStaking::addToPosition` should assing the amount multiplier based on the new lock duration instead of initial lock duration.

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/138

Found by

Honour, Naresh, PUSH0, Smacaud, dany.armstrong90, dhank, iamnmt, jah, karsar, neon2835, qmdddd, robertodf, sheep, walter

Summary

There are two separate issues that make necessary to assign the multiplier based on the new lock duration:

1. First, when users add tokens to their position via `MlumStaking::addToPosition`, the new remaining time for the lock duration is recalculated as the amount-weighted lock duration. However, when the remaining time for an existing deposit is 0, this term is omitted, allowing users to retain the same amount multiplier with a reduced lock time. Consider the following sequence of actions:
    - Alice creates a position by calling `MlumStaking::createPosition`depositing 1 ether and a lock time of 365 days
    - After the 365 days elapse, Alice adds another 1 ether to her position. The snippet below illustrates how the new lock time for the position is calculated:

```
uint256 avgDuration = (remainingLockTime *
    position.amount +
    amountToAdd *
    position.initialLockDuration) / (position.amount + amountToAdd);
position.startLockTime = _currentBlockTimestamp();
position.lockDuration = avgDuration;

// lock multiplier stays the same
position.lockMultiplier = getMultiplierByLockDuration(
    position.initialLockDuration
);
```

    - The result will be: `(0*1 ether + 1 ether*365 days)/ 2 ether`, therefore Alice will need to wait just half a year, while the multiplier remains unchanged.

2. Second, the missalignment between this function and `MlumStaking::renewLockPosition` creates an arbitrage opportunity for users, allowing them to reassign the lock multiplier to the initial duration if it is more beneficial. Consider the following scenario:
    - Alice creates a position with an initial lock duration of 365 days. The multiplier will be 3.
    - Then after 9 months, the lock duration is updated, let's say she adds to the position just 1 wei. The new lock duration is 90 days.
    - After another 30 days, she wants to renew her position for another 90 days. Then she calls `MlumStaking::renewLockPosition`. The new amount multiplier will be calculated as `1+90/365*2 < 3`.
    - Since it is not in her interest to have a lower multiplier than originally, then she adds just 1 wei to her position. The new multiplier will be 3 again.

Vulnerability Detail

Impact

You may find below the coded PoC corresponding to each of the aforementioned scenarios:

```
function testLockDurationReduced() public {
    _stakingToken.mint(ALICE, 2 ether);

    vm.startPrank(ALICE);
    _stakingToken.approve(address(_pool), 1 ether);
    _pool.createPosition(1 ether, 365 days);
    vm.stopPrank();

    // check lockduration
    MlumStaking.StakingPosition memory position = _pool.getStakingPosition(
        1
    );
    assertEq(position.lockDuration, 365 days);

    skip(365 days);

    // add to position should take calc. avg. lock duration
    vm.startPrank(ALICE);
    _stakingToken.approve(address(_pool), 1 ether);
    _pool.addToPosition(1, 1 ether);
    vm.stopPrank();

    position = _pool.getStakingPosition(1);
```


SHERLOCK

```
    assertEq(position.lockDuration, 365 days / 2);
    assertEq(position.amountWithMultiplier, 2 ether * 3);
}
```

```
function testExtendLockAndAdd() public {
    _stakingToken.mint(ALICE, 2 ether);

    vm.startPrank(ALICE);
    _stakingToken.approve(address(_pool), 2 ether);
    _pool.createPosition(1 ether, 365 days);
    vm.stopPrank();

    // check lockduration
    MlumStaking.StakingPosition memory position = _pool.getStakingPosition(
        1
    );
    assertEq(position.lockDuration, 365 days);

    skip(365 days - 90 days);
    vm.startPrank(ALICE);
    _pool.addToPosition(1, 1 wei); // lock duration  90 days
    skip(30 days);
    _pool.renewLockPosition(1); // multiplier  1+90/365*2
    position = _pool.getStakingPosition(1);
    assertEq(position.lockDuration, 7776000);
    assertEq(position.amountWithMultiplier, 14931000000000000001);

    _pool.addToPosition(1, 1 wei); // multiplier = 3
    position = _pool.getStakingPosition(1);
    assertEq(position.lockDuration, 7776000);
    assertEq(position.amountWithMultiplier, 30000000000000000006);
}
```

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/MlumStaking.sol#L409-L417

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/MlumStaking.sol#L509-L514

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/MlumStaking.sol#L714

Tool used

Manual Review

Recommendation

Assign new multiplier in `MlumStaking::addToPosition` based on lock duration rather than initial lock duration.

Discussion

0xSmartContract

This issue causes users to keep the same multiplier by reducing lock times, thus gaining an advantage.

This creates an unfair advantage in the system and undermines the reliability of the staking mechanism.

0xHans1

PR: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/138 Fixes scenario 2.

For scenario 1: it is a design choice that the amount mutliplier stays the same even lock duration ended.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/5

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-3: Adding genuine BribeRewarder contract instances to a pool in order to incentivize users can be DOSed

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/190

Found by

0xAnmol, 0xAsen, 0xR360, DMoore, Honour, KupiaSec, MrCrowNFT, PUSH0, Reentrants, Silvermist, anonymousjoe, araj, aslanbek, blockchain555, coffiasd, dany.armstrong90, dimulski, gkrastenov, iamnmt, jennifer37, kmXAdam, oualidpro, radin200, rsam_eth, sajjad, santipu_, scammed, slowfi, tedox, utsav, web3pwn, zarkk01

Summary

In the `Voter.sol` contract, protocols can create `BribeRewarder.sol` contract instances in order to bribe users to vote for the pool specified by the protocol. The more users vote for a specific pool, the bigger the weight of that pool will be compared to other pools and thus users staking the pool **LP** token in the `MasterchefV2.sol` contract will receive more rewards. The LP token of the pool the protocol is bribing for, receives bigger allocation of the **LUM** token in the `MasterchefV2.sol` contract. Thus incentivizing people to deposit tokens in the AMM associated with the LP token in order to acquire the LP token for a pool with higher weight, thus providing more liquidity in a trading pair. In the `Voter.sol` contract a voting period is defined by an id, start time, and a end time which is the start time + the globaly specified **_periodDuration**. When a protocol tries to add a `BribeRewarder.sol` contract instance for a pool and a voting period, they have to call the fundAndBribe() function or the bribe() funciton and supply the required amount of reward tokens. Both of the above mentioned functions internally call the _bribe() function which in turn calls the onRegister() function in the `Voter.sol` contract. There is the following check in the onRegister() function:

```
function onRegister() external override {
    IBribeRewarder rewarder = IBribeRewarder(msg.sender);

    _checkRegisterCaller(rewarder);

    uint256 currentPeriodId = _currentVotingPeriodId;
    (address pool, uint256[] memory periods) = rewarder.getBribePeriods();
    for (uint256 i = 0; i < periods.length; ++i) {
        // TODO check if rewarder token + pool  is already registered

        require(periods[i] >= currentPeriodId, "wrong period");
```


SHERLOCK

```
        require(_bribesPerPriod[periods[i]][pool].length + 1 <=
↪   Constants.MAX_BRIBES_PER_POOL, "too much bribes");
        _bribesPerPriod[periods[i]][pool].push(rewarder);
    }
}
```

Constants.MAX_BRIBES_PER_POOL is equal to 5. This means that each pool can be associated with a maximum of 5 instances of the `BribeRewarder.sol` contract for each voting period. The problem is that adding `BribeRewarder.sol` contract instances for a pool for a certain voting period is permissionless. There is no whitelist for the tokens that can be used as reward tokens in the `BribeRewarder.sol` contract or a minimum amount of rewards that have to be distributed per each voting period. A malicious actor can just deploy an ERC20 token on the network, that have absolutely no dollar value, mint as many tokens as he wants and then deploy 5 instance of the `BribeRewarder.sol` contract by calling the createBribeRewarder() function in the `RewarderFactory.sol` contract. After that he can either call fundAndBribe() function or the bribe() function in order to associate the above mentioned `BribeRewarder.sol` contract instances for a specific pool and voting period. A malicious actor can specify as many voting periods as he want. _periodDuration is set to 1209600 seconds = 2 weeks on `Voter.sol` initialization. If a malicious actor sets `BribeRewarder.sol` contract instances for 100 voting periods, that means 200 weeks or almost 4 years, no other `BribeRewarder.sol` contract instance can be added during this period. When real rewards can't be used as incentives for users, nobody will vote for a specific pool. A malicious actor can dos all pools that can potentially be added for example by tracking what new pools are created at the MagicSea exchange and then immediately performing the above specified steps, thus making the entire `Voter.sol` and `BribeRewarders.sol` contracts obsolete, as their main purpose is to incentivize users to vote for specific pools and reward them with tokens for their vote. Or a malicious actor can dos specific projects that are direct competitors of his project, thus more people will provide liquidity for an AMM he is bribing users for, this way he can also provide much less rewards and still his pool will be the most lucrative one.

Vulnerability Detail

Gist

After following the steps in the above mentioned gist add the following test to the `AuditorTests.t.sol` file:

```
function test_BrickRewardBribers() public {
    vm.startPrank(attacker);
    customNoValueToken.mint(attacker, 500_000e18);
    BribeRewarder rewarder = BribeRewarder(payable(address(rewarderFactory.creat
↪   eBribeRewarder(customNoValueToken, pool))));
```


SHERLOCK

```
    customNoValueToken.approve(address(rewarder), type(uint256).max);
    rewarder.fundAndBribe(1, 100, 100e18);

    BribeRewarder rewarder1 = BribeRewarder(payable(address(rewarderFactory.crea
↪   teBribeRewarder(customNoValueToken, pool))));
    customNoValueToken.approve(address(rewarder1), type(uint256).max);
    rewarder1.fundAndBribe(1, 100, 100e18);

    BribeRewarder rewarder2 = BribeRewarder(payable(address(rewarderFactory.crea
↪   teBribeRewarder(customNoValueToken, pool))));
    customNoValueToken.approve(address(rewarder2), type(uint256).max);
    rewarder2.fundAndBribe(1, 100, 100e18);

    BribeRewarder rewarder3 = BribeRewarder(payable(address(rewarderFactory.crea
↪   teBribeRewarder(customNoValueToken, pool))));
    customNoValueToken.approve(address(rewarder3), type(uint256).max);
    rewarder3.fundAndBribe(1, 100, 100e18);

    BribeRewarder rewarder4 = BribeRewarder(payable(address(rewarderFactory.crea
↪   teBribeRewarder(customNoValueToken, pool))));
    customNoValueToken.approve(address(rewarder4), type(uint256).max);
    rewarder4.fundAndBribe(1, 100, 100e18);
    vm.stopPrank();

    vm.startPrank(tom);
    BribeRewarder rewarderReal = BribeRewarder(payable(address(rewarderFactory.c
↪   reateBribeRewarder(bribeRewardToken, pool))));
    bribeRewardToken.mint(tom, 100_000e6);
    bribeRewardToken.approve(address(rewarderReal), type(uint256).max);
    customNoValueToken.approve(address(rewarderReal), type(uint256).max);
    vm.expectRevert(bytes("too much bribes"));
    rewarderReal.fundAndBribe(2, 6, 20_000e6);
    vm.stopPrank();
}
```

To run the test use: `forge test -vvv --mt test_BrickRewardBribers`

Impact

A malicious actor can dos the adding of `BribeRewarder.sol` contract instances for specific pools, and thus either make `Voter.sol` and `BribeRewarders.sol` contracts obsolete, or prevent the addition of genuine `BribeRewarder.sol` contract instances for a specific project which he sees as a competitor. I believe this vulnerability is of high severity as it severely restricts the availability of the main functionality of the `Voter.sol` and `BribeRewarders.sol` contracts.

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/Voter.sol#L130-L144

Tool used

Manual Review & Foundry

Recommendation

Consider creating a functionality that whitelists only previously verified owner of pools, or people that really intend to distribute rewards to voters, and allow only them to add `BribeRewarders.sol` contracts instances for the pool they are whitelisted for.

Discussion

0xSmartContract

In `Voter.sol` and `BribeRewarders.sol` contracts, a maximum of 5 `BribeRewarder.sol` contracts are allowed per pool and these can be added without permission. A malicious actor could use worthless tokens to fill this limit, preventing real rewards from being added. This results in user incentives being ineffective and the voting system becoming dysfunctional.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/35

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-4: Rewards might get stuck when approved actor renews a position

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/207

Found by

0xAnmol, BlockBusters, ChinmayF, HonorLt, ydlee

Summary

When an approved actor calls the harvest function, the rewards get sent to the user (staker). However, when the approved actor renews the user's position, they receive the rewards instead.

If the approved actor is a smart contract (e.g., a keeper), the funds might get stuck forever or go to the wrong user, such as a Chainlink keeper.

Vulnerability Detail

Suppose Alice mints an NFT by creating a position and approves Bob to use it.

- When Bob calls `harvestPosition` with Alice's `tokenId`, Alice will receive the rewards (as intended)
- When Bob calls `renewLockPosition` with Alice's `tokenId`, Bob will receive the rewards. The internal function `_lockPosition`, which is called by `renewLockPosition`, also harvests the position before updating the lock duration. Unlike the harvest function, `_lockPosition` sends the rewards to msg.sender instead of the token owner.

This bug exists in both `renewLockPosition` and `extendLockPosition`, as they both call `_lockPosition`, which includes the wrong receiver.

PoC

To run this test, add it into `MlumStaking.t.sol`.

```
function testVuln_ApprovedActorReceivesRewardsWhenRenewingPosition() public {
    // setup pool
    uint256 _amount = 100e18;
    uint256 lockTime = 1 days;

    _rewardToken.mint(address(_pool), 100_000e6);
    _stakingToken.mint(ALICE, _amount);
```

```
    // alice creates new position
    vm.startPrank(ALICE);
    _stakingToken.approve(address(_pool), _amount);
    _pool.createPosition(_amount, lockTime);
    vm.stopPrank();

    // alice approves bob
    vm.prank(ALICE);
    _pool.approve(BOB, 1);

    skip(1 hours);

    // for simplicity of the PoC we use a static call
    // IMlumStaking doesn't include `renewLockPosition(uint256)`
    uint256 bobBefore = _rewardToken.balanceOf(BOB);
    vm.prank(BOB);
    address(_pool).call(abi.encodeWithSignature("renewLockPosition(uint256)",
↪   1));

    // Bob receivew the rewards, instead of alice
    assertGt(_rewardToken.balanceOf(BOB), bobBefore);
}
```

Impact

Possible loss of reward tokens

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/MlumStaking.sol#L710

Tool used

Manual Review, Foundry

Recommendation

Change `_lockPosition()` in `MlumStaking.sol` to use the owner of the position instead of `msg.sender`.

```
function _lockPosition(uint256 tokenId, uint256 lockDuration, bool resetInitial)
↪   internal {
    ...
-   _harvestPosition(tokenId, msg.sender);
```


SHERLOCK

```
+    _harvestPosition(tokenId, _ownerOf(tokenId));
     ...
}
```

Discussion

0xSmartContract

Calling the `renewLockPosition` and `extendLockPosition` functions may result in rewards being sent to the wrong address.

When an approved actor calls the `renewLockPosition` or `extendLockPosition` functions, the rewards go to the approved actor who made the call, not the Position Owner.

0xHans1

PR: https://github.com/metropolis-exchange/magicsea-staking/pull/8

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/8

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-5: Voting and bribe rewards can be hijacked during emergency unlock by already existing positions

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/290

Found by

ChinmayF, KupiaSec, araj, aslanbek, chaduke, kmXAdam, robertodf

Summary

There are many ways of modifying an already existing staking position : but they exhibit different behavior when an emergency unlock is made active. `renewLock()` and `extendLock()` are straight away disallowed. `withdrawFromPosition()` and `emergencyWithdraw()` are allowed but it can only pull out assets and not do anything else.

`addToPosition()` is a different case, it has no restrictions even when the emergency unlock is active. This can be used by an attacker to modify pre-existing position to vote with a new amount in Voter.

This is a big problem because all other ways of starting, or renewing a staking position are blocked.

Vulnerability Detail

If a position already exists before the emergency unlock started, the owner of such positions can add amounts to them, and use this new amount for voting in the current voting period.

All other existing positions can also vote, but no new amount of assets can enter the voting circle because :

- calling createPosition will set lockDuration to 0 for this new position (which will revert when voting due to being < minimumLockTime) : see here and here
- calling renewLock and extendLock is straightaway reverted.

Now because no new assets can enter the voting amount circulation during this period, this creates a monopoly for already existing positions to control the voting outcome with lesser participants on board.

This can be particularly used by an attacker to hijack the voting process in the following steps :

- If the attacker already knows the peculiarty of `addToPosition()`, they can plan in advance and open many many 1 wei staking positions with long lock

duration (such that they are eligible for voting based on minimumLockTime etc. in Voter.sol) and wait for the emergency unlock.

- If the emergency unlock happens they suddenly start adding large amounts to their 1 wei positions.
- This increases their amountWithMultiplier(sure multiplier will get limited to 1x but we are talking about the amount itself) and thus they have more voting power.
- emergencyUnlock is meant to be used in an emergency which can happen anytime. If a voting period is running during that time, there is no way to stop the current voting period.
- The attacker can use this new voting power for all their positions to hijack the voting numbers because other users cant open new positions or renew/extend, and honest users are not aware of this loophole in the system via addToPosition()

In this way, an attacker can gain monopoly over the votes during an emergencyUnlock, which ultimately influences the LUM emissions directed to certain pools.

Impact

Voting process can be manipulated during an emergencyUnlock using pre-existing positions. This also leads to a hijack of the bribe rewards if any bribe rewarders are set for those pools.

High severity because this can be used to unfairly influence the voting outcome and misdirect LUM rewards, which is the most critical property of the system. Also this is unfair to all other voters as they lose out on bribe rewards, and the attacker can get a very high share of it.

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/MlumStaking.sol#L397

Tool used

Manual Review

Recommendation

This can be solved by simply disallowing any activity apart from emregencyWithdraw during the time when emergencyUnlock is active. Apply the

check used in _lockPosition() to addToPosition() too.

Discussion

0xSmartContract

During emergency unlock, adding to existing positions can increase voting power and manipulate the voting process. This can unfairly affect voting outcomes and rewards.

0xHans1

PR: https://github.com/metropolis-exchange/magicsea-staking/pull/10

0xSmartContract

These issues all focus on how the system can be abused during emergency unlocking, which can have negative consequences for users and the contract. The main goal is to ensure that only safe and fair transactions are made during emergency unlocking, and to prevent malicious users from exploiting system vulnerabilities to gain unfair advantage.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/10

chinmay-farkya

Escalate

There are many duplication errors here, out of which most issues are invalid in reality. These are listed below:

- #15 is not a duplicate because it simply says addToPosition should have a certain check without describing the issue created. The impact has not been included, neither does the submission describe a valid attack path. This is against sherlock's duplication rules here
- #131 the core point is the decrease in user's rewards which is invalid because the emergencyWithdraw() function has comments saying the rewards have been let go by the staker because its an emergency, in which case the rewards he lost will not even be redistributed as pointed out by issue #460. So the new staker cannot gain the rewards left out by emergency withdrawers as per current logic. This is invalid and not a duplicate.
- #134 is not a duplicate. Its invalid because frontrunning is not possible on Iota evm and the contest readme says the protocol is to be deployed on Iota only. See here : IOTA VM Docs

- #184 has wrong recommendation because avgDuration needs to be actually increased generally, the impact mentioned there is also wrong. The ability to increase lockDuration is an expected behavior (thats how the design is) and becoming eligible for voting has nothing to do with emergencyunlock, it can also be done in normal operations and is not malicious. This doesn't mention the increase in voting power which is the primary issue, voting eligibility is of no concern. Hence invalid.
- #387 again lacks any valid attack path and has no description of any impact. According to sherlock rules, a valid duplicate should “Identify a valid attack path or vulnerability path”. Hence invalid
- #567 is completely invalid. The description of the issue says ”So no matter how long the user staked before the current amountWithMultiplier will be amount multiplied with 1 instead of the right multiplier which will affect the reward” which is false because before the lockmultiplier gets updated, _harvestPosition() is called which uses the previous amountWithMultiplier to calculate the rewards earned uptil now and directly sends these rewards to the user. After that multiplier is updated for the position and that is expected behavior as the previous rewards have already been claimed and emergency unlock time period is meant for withdrawing and not to earn multiplied rewards for the period itself.
- #575 is invalid. The core point is that users can use withdrawFromPosition during emergency unlock. and claim rewards they had earned. But this is expected behavior. The rewards they earn were anyways earned by their positions, so its not a malicious action and does not lead to an attack or loss of funds for anyone.
- #660 is invalid due to the same reasoning as #575. Users using withdrawFromPosition during emergencyUnlock is completely normal as they only get their own rewards which they deserved.
- #368 is not a duplicate of this issue, but a duplicate of #460.

Apart from these listed, other issues seem to be valid duplicates of #290.

sherlock-admin3

> Escalate
>
> There are many duplication errors here, out of which most issues are invalid in reality. These are listed below:
>
> - #15 is not a duplicate because it simply says addToPosition should have a certain check without describing the issue created. The impact has not been included, neither does the submission describe a valid attack path. This is against sherlock’s duplication rules here

- #131 the core point is the decrease in user's rewards which is invalid because the emergencyWithdraw() function has comments saying the rewards have been let go by the staker because its an emergency, in which case the rewards he lost will not even be redistributed as pointed out by issue #460. So the new staker cannot gain the rewards left out by emergency withdrawers as per current logic. This is invalid and not a duplicate.
- #134 is not a duplicate. Its invalid because frontrunning is not possible on Iota evm and the contest readme says the protocol is to be deployed on Iota only. See here : IOTA VM Docs
- #184 has wrong recommendation because avgDuration needs to be actually increased generally, the impact mentioned there is also wrong. The ability to increase lockDuration is an expected behavior (thats how the design is) and becoming eligible for voting has nothing to do with emergencyunlock, it can also be done in normal operations and is not malicious. This doesn't mention the increase in voting power which is the primary issue, voting eligibility is of no concern. Hence invalid.
- #387 again lacks any valid attack path and has no description of any impact. According to sherlock rules, a valid duplicate should "Identify a valid attack path or vulnerability path". Hence invalid
- #567 is completely invalid. The description of the issue says "So no matter how long the user staked before the current amountWithMultiplier will be amount multiplied with 1 instead of the right multiplier which will affect the reward" which is false because before the lockmultiplier gets updated, _harvestPosition() is called which uses the previous amountWithMultiplier to calculate the rewards earned uptil now and directly sends these rewards to the user. After that multiplier is updated for the position and that is expected behavior as the previous rewards have already been claimed and emergency unlock time period is meant for withdrawing and not to earn multiplied rewards for the period itself.
- #575 is invalid. The core point is that users can use withdrawFromPosition during emergency unlock. and claim rewards they had earned. But this is expected behavior. The rewards they earn were anyways earned by their positions, so its not a malicious action and does not lead to an attack or loss of funds for anyone.
- #660 is invalid due to the same reasoning as #575. Users using withdrawFromPosition during emergencyUnlock is completely normal as they only get their own rewards which they deserved.

- #368 is not a duplicate of this issue, but a duplicate of #460.

Apart from these listed, other issues seem to be valid duplicates of #290.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xAraj

Totally agree with @chinmay-farkya, above issue is about how an attacker can take advantage of addToPosition() while emergency to vote for a malicious pool. Not all above tagged issue describe the same attack path

J4X-98

Hey @chinmay-farkya @0xSmartContract ,

#15 clearly describes precisely the issue of the main submission. Being able to add liquidity on a paused contract is by itself already a medium impact. While the main issue described another medium impact (that leverages this), #15 should stay a valid dup.

Oot2k

I think this issue should be grouped with #138 or low.

Both have the same root cause of: addToPosition allowing to higher voting power. Fixing 138 would fix this issue as well, because if addToPosition would update the times correctly it would not be possible to vote anymore.

I would also want to point out that this issue describes an extreme edge case Szenario:

- One bribe rewarder only rewards one Staking contract
- In case the admin decides to active emergency withdraw, all reward tokens are considered lost by definition (only way to withdraw is without claiming them)
- so even in case a user is abled to vote for reward distribution, the outcome of this vote does not matter

Hench I would consider the impact of this issue to be low.

chinmay-farkya

@J4X-98

> https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/15 clearly describes precisely the issue of the main submission. Being able to add liquidity on a paused contract is by itself already a medium

> impact. While the main issue described another medium impact (that leverages this), https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/15 should stay a valid dup.

Your issue 15 does not identify any valid attack path. Simply stating that liquidity can be added while paused cannot be considered as a medium impact unless it can be shown to have second order effects or a loss/manipulation of some kind.

This issue 290 describes such an impact of misuse of the voting power while 15 does not identify it.

Pascal4me

@0xSmartContract My issue #567 even though it's not a duplicate doesn't mean it is invalid. It addresses the ability of a user to alter their position when contract is unlocked due to lack of access control on 'addToPosition()', yes the user will harvest current rewards with correct multiplier then lock multiplier will be set to zero because pool is unlocked so if the user comes back in let's say 6 months time to harvest again they'll get rewards calculates with zero lock multiplier. But they all kinda have same root cause so I don't know @WangSecurity I clearly started the issue for validity here

chinmay-farkya

> I think this issue should be grouped with #138 or low.
>
> Both have the same root cause of: addToPosition allowing to higher voting power. Fixing 138 would fix this issue as well, because if addToPosition would update the times correctly it would not be possible to vote anymore.
>
> I would also want to point out that this issue describes an extreme edge case Szenario:
>
> - One bribe rewarder only rewards one Staking contract
> - In case the admin decides to active emergency withdraw, all reward tokens are considered lost by definition (only way to withdraw is without claiming them)
> - so even in case a user is abled to vote for reward distribution, the outcome of this vote does not matter
>
> Hench I would consider the impact of this issue to be low.

The assertion that this issue is same as #138 is incorrect because #138 is talking about how you can keep the position active with the same multiplier(and thus gain unfair share of rewards) under "normal operations of the system" due to the use of initialLockDuration, while this issue 290 talks about how during "only an

emergencyUnlock" (and not under normal operations like #138) the positions can be gamed to immediately gain extra voting power and manipulate the voting data.

Both have separate root causes : core issue of 138 is that initialLockDuration is used for assigning multiplier "under normal operation" while core issue of 290 is : during emergency unlock, addToPosition is allowed to add amount and gain greater voting power

Both have separate fixes : fix for 138 is : to use lockDuration while assigning multiplier under all "normal operations" fix for 290 is : restrict calling addToPosition while emergency unlock is active

These are clearly two separate issues.

bbl4de

> @J4X-98
>
> > #15 clearly describes precisely the issue of the main submission. Being able to add liquidity on a paused contract is by itself already a medium impact. While the main issue described another medium impact (that leverages this), #15 should stay a valid dup.
>
> Your issue 15 does not identify any valid attack path. Simply stating that liquidity can be added while paused cannot be considered as a medium impact unless it can be shown to have second order effects or a loss/manipulation of some kind.
>
> This issue 290 describes such an impact of misuse of the voting power while 15 does not identify it.

I agree with @J4X-98 regarding the validity of issues based simply on the possibility to add liquidity to a paused contract. Also, issue #131 describes this impact which can be mitigated in the same way as the main issue. Your report - #290 is indeed presenting a valid attack vector that is possible *based* on the impact described by #131. I simply did not state what the malicious could do with the additional voting power. It is not a valid argument for invalidating an issue, rather a justification for the main issue to be chosen for the report.

chinmay-farkya

> It is not a valid argument for invalidating an issue, rather a justification for the main issue to be chosen for the report.

Well we are governed by sherlock's rules for valid duplications. Its not a justification for an issue getting chosen for the report, but for what counts as a valid duplicate and what does not.

> The duplication rules assume we have a "target issue", and the "potential

> duplicate" of that issue needs to meet the following requirements to be considered a duplicate. Identify the root cause Identify at least a Medium impact Identify a valid attack path or vulnerability path Only when the "potential duplicate" meets all four requirements will the "potential duplicate" be duplicated with the "target issue", and all duplicates will be awarded the highest severity identified among the duplicates.

bbl4de

> > It is not a valid argument for invalidating an issue, rather a justification for the main issue to be chosen for the report.
>
> Well we are governed by sherlock's rules for valid duplications. Its not a justification for an issue getting chosen for the report, but for what counts as a valid duplicate and what does not.
>
> > The duplication rules assume we have a "target issue", and the "potential duplicate" of that issue needs to meet the following requirements to be considered a duplicate. Identify the root cause Identify at least a Medium impact Identify a valid attack path or vulnerability path Only when the "potential duplicate" meets all four requirements will the "potential duplicate" be duplicated with the "target issue", and all duplicates will be awarded the highest severity identified among the duplicates.

You're right. I should reiterate my reply. Adding liquidity to a paused contract is an attack vector - it's very straight forward because the bug is quite simple. What you have presented is additional **impact** of the attack vector. With this in mind, we may go through the requirements for a valid duplicate from the rules you have cited for issue #131: 1.Root cause - insufficient validation *included* 2.Medium impact - adding liquidity to a paused contract, manipulating reward distribution *included* 3.Valid attack path - simply calling `addToPosition()` at the right time is the attack vector in itself *included*

My #131 finding is therefore a valid duplicate according to the rules. What you provided is additional impact based on the root cause and primary impact, as @J4X-98 commented.

Note that I'm referring to my issue #131, I did not review other escalated findings.

J4X-98

> @J4X-98
>
> > https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/15 clearly describes precisely the issue of the main submission. Being able to add liquidity on a paused contract is by itself already a medium impact. While the main issue described another medium impact (that leverages this),

> https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/15 should stay a valid dup.
>
> Your issue 15 does not identify any valid attack path. Simply stating that liquidity can be added while paused cannot be considered as a medium impact unless it can be shown to have second order effects or a loss/manipulation of some kind.
>
> This issue 290 describes such an impact of misuse of the voting power while 15 does not identify it.

Pausing is a security feature (like for example blacklisting) which is clearly bypassed and that's also described in our issue. This is medium on any platform.

Just because you found another medium impact it doesn't mean that all other medium impacts resulting out of the same root cause are invalid.

yash-0025

Issue #660 also shows that the rewards fund is stucked for the user during emergency withdraw with the POC screenshot in the 2nd scenario and the same issue which depicts the same Issue #460 is considered valid, So isn't it a mistaken duped rather than an invalid.

chinmay-farkya

Well @J4X-98 now I'm curious to know when did "adding liquidity while paused" become a medium impact in itself.

I disagree because if the liquidity they are adding during a paused state can be withdrawn later, and as long as it does not affect anything else in the system (like rewards etc.), it is not a medium impact but a low one.

J4X-98

Pausing means the protocol is paused e.g. no one can add or withdraw liquidity. The same way it is in any other DEX, vault etc. that's the whole reasoning behind it. This is broken by the implementation.

I don't think discussing with you makes any sense if this is not clearly a medium to you. I've already added my info for the judge and will wait for his judging.

0xSmartContract

This issue can be grouped with #138

chinmay-farkya

@0xSmartContract

This and #138 are completely different issues

#138 :

- It is about how the lockDuration calculation can be skewed and misused
- Bug manifests in normal operations
- Issue is that a certain multiplier effect can be maintained for longer than deserved
- Fix is to use new lockDuration when calculating multiplier

#290 :

- It is not related to the lockDuration, it is about how a position's vote amount can be increased
- Bug will only manifest in emergency unlock situation, and not under normal operations
- In case of emergency unlock, the staker actually loses the multiplier instead of maintaining the previous one, the new multiplier is zero
- Fix is to not allow addToPosition when emergencyUnlock is active.

0xSmartContract

This issue can be grouped with #138

They have similar root causes: Both issues are related to the position update logic in the MlumStaking contract. Issue 138 is related to incorrect multiplier calculation when adding to positions, while issue 290 is related to adding to positions during emergency locks.

They have interrelated effects: Both issues can affect users' voting power and rewards. 138 allows users to get higher multipliers, while 290 allows voting manipulation during emergency locks.

Overlapping of solutions: To solve both issues, it is necessary to change the behavior of the addToPosition() function. These changes can be considered together to provide a more comprehensive solution.

Both issues are related to the position management and locking logic in the MlumStaking contract.

For these reasons, combining issues 138 and 290 can provide a more holistic and efficient solution, and it is a nice duplicate of these two to improve the overall code quality.

We are hearing from only @chinmay-farkya about grouped with 138 , this is not healthy, I would like to get comments from many valid Watsons on this issue

J4X-98

I think grouping these issues is not the right way ahead. Our issue #15 describes that pausing does not correctly work which is in no way related to #138. Grouping

should be done based on the root cause not based on which issues affect a similar part of the code. Just because there are separate issues in the same function, they can not just be grouped into one.

0xSmartContract

1- At the moment we need to agree on now is: (This issue can be grouped with #138) 2-Then we look at the status of duplicates (incl. #15)

J4X-98

As stated before this group of issues and #138 are completely different issues. They just occur in a similar part of the code but have different root causes, different scenarios and different mitigations.

chinmay-farkya

@0xSmartContract I don't know why you are saying that this is similar to #138 . These are two completely different issues with different fixes, root causes, and affecting different mechanisms.

One is about lockDuration and multipliers, the other (this #290) is about manipulating the vote amount via addPOsition instead. It is not even remotely related to that as pointed in my comment here. Bugs manifest in completely different situations (the multiplier issue #138 will not even happen in case of emergency Unlocks but the lock amount s can only be manipulated in case of emergencyUnlock).

> Overlapping of solutions: To solve both issues, it is necessary to change the behavior of the addToPosition() function

While the protocol team may need to comprehensively look at all bugs together, that doesn't mean that the issues are duplicates. While fixing any issue, the team will always need to look at the fix's effects on the whole system, does that make all the issues of a contract under one duplicate ?

rdf5

Here I agree with @chinmay-farkya regarding the differences between both issues. While both stem from a malfunction in the same function, they can be considered independent of each other.

Issue #138 involves incorrect staking logic, whereas issue #290 pertains to unrestricted access control. The solution for the former involves changing the multiplier calculation logic, while the latter requires implementing a proper access control mechanism.

The consequences of the malfunction and the solutions for each issue are not interconnected. Unless both solutions can be addressed with a single approach, I wouldn’t consider them related.

WangSecurity

About issues mentioned in the escalation comment:

1. #15 -- I agree it's a different issue, moreover, it looks as a design recommendation, just being able to deposit when you shouldn't I believe is not an impact but rather the root cause. Plus, I don't see any information that there should be such a check, hence, it may be intended.
2. #131 -- also not a duplicate and looks like describing the intended functionality.
3. #134 -- also not a duplicate. But I would argue that currently, front-running is still possible on IOTA. It's unreliable, but you still can see the mempool (until the new IOTA EVM is launched with no mempool at the end of 2024), so you can send lots of transactions increasing the chance of successful front-running. But, this also requires the reward token to be minted right before the emergency unlock is set to true, moreover, looks to be intended. Hence, I agree with it being invalid.
4. #184 -- as I understand, the escalation comment is precise here, agree it's invalid.
5. #387 -- don't see any medium severity impact.
6. #567 -- again, the escalation comment is precise here, will invalidate it.
7. #575 -- don't see a medium impact and seems to be intended.
8. #660 -- same as above
9. #368 -- agree it's a duplicate of #460

I agree it's not a duplicate of #138 based on this, this and this comments. Let me know if you want a deeper analysis of these 2 issues, but I don't see a point in me repeating the same arguments another time.

Planning to accept the escalation, invalidate 8 issues above and keep this a separate high severity family.

WangSecurity

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- chinmay-farkya: accepted

chinmay-farkya

SHERLOCK

Hey @WangSecurity did you intend to keep this as high severity ?

Your previous comment says this :

> keep this a separate high severity family

WangSecurity

Yes, you're correct, excuse me for the typo, I meant to keep this as medium as it was initially. But if you have arguments disputing the severity, please share them.

chinmay-farkya

I think it is medium, so we'll settle it down.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-6: Inconsistent check in `harvestPositionsTo()` function

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/329

Found by

AuditorPraise, BengalCatBalu, ChinmayF, Honour, Reentrants, Ryonen, bbl4de, blackhole, coffiasd, dany.armstrong90, minhquanym, novaman33, radin200, scammed, sh0velware

Summary

Inconsistent check in `harvestPositionsTo()` function limits the ability of approved address to harvest on behalf of owner.

Vulnerability Detail

In the function `harvestPositionsTo()`, function `_requireOnlyApprovedOrOwnerOf()` allows owner or approved address to harvest for the position.

However, the check `(msg.sender == tokenOwner && msg.sender == to)` only allowing the caller to be token owner. Thus these 2 checks are contradicted.

```
function harvestPositionsTo(uint256[] calldata tokenIds, address to) external
↪   override nonReentrant {
    _updatePool();

    uint256 length = tokenIds.length;

    for (uint256 i = 0; i < length; ++i) {
        uint256 tokenId = tokenIds[i];
        _requireOnlyApprovedOrOwnerOf(tokenId);
        address tokenOwner = ERC721Upgradeable.ownerOf(tokenId);
        // if sender is the current owner, must also be the harvest dst address
        // if sender is approved, current owner must be a contract
        // @audit not consistent with _requireOnlyApprovedOrOwnerOf()
        require(
            (msg.sender == tokenOwner && msg.sender == to), // legacy ||
↪   tokenOwner.isContract()
            "FORBIDDEN"
        );

        _harvestPosition(tokenId, to);
        _updateBoostMultiplierInfoAndRewardDebt(_stakingPositions[tokenId]);
```


SHERLOCK

```
    }
}
```

Impact

Contradictions in the function `harvestPositionsTo()`. Approved address cannot call `harvestPositionsTo()` on behalf of NFT owner.

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/MlumStaking.sol#L475-L484

Tool used

Manual Review

Recommendation

The intended check in function `harvestPositionsTo()` might be, changing `&&` to `||`

```
require(
-     (msg.sender == tokenOwner && msg.sender == to), // legacy ||
 ↪  tokenOwner.isContract()
+     (msg.sender == tokenOwner || msg.sender == to), // legacy ||
 ↪  tokenOwner.isContract()
    "FORBIDDEN"
);
```

Discussion

0xSmartContract

Due to conflicting checks in the `harvestPositionsTo()` function, authorized addresses are prevented from harvesting on behalf of the NFT owner.

0xHans1

This was fixed during an other audit. add code comment to show the fix in the PR

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/23

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-7: Attacker can manipulate the `lockDuration` of other users positions

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/378

Found by

.-..---.....-., 0uts1der, 0xAsen, 0xboriskataa, Aymen0909, BengalCatBalu, ChinmayF, Gowtham_Ponnana, HonorLt, Honour, KupiaSec, PNS, PUSH0, PeterSR, Ryonen, Silvermist, TessKimy, Yashar, aslanbek, bbl4de, blackhole, coffiasd, dev0cloo, dhank, eLSeR17, fibonacci, gkrastenov, iamnmt, jennifer37, kmXAdam, luke, minhquanym, neogranicen, neon2835, nikhil840096, radin200, rsam_eth, scammed, sh0velware, slowfi, t.aksoy, web3pwn, zarkk01

Summary

The `_requireOnlyOperatorOrOwnerOf` check in `addToPosition` can be bypassed, allowing attackers to manipulate the `lockDuration` of other users positions, impacting voting eligibility and preventing users from participating in voting and earning rewards.

Vulnerability Detail

The _requireOnlyOperatorOrOwnerOf check in `addToPosition` ensures that the caller is either the owner or the operator of the tokenId:

```
function addToPosition(uint256 tokenId, uint256 amountToAdd) external override
↪  nonReentrant {
    _requireOnlyOperatorOrOwnerOf(tokenId);
```

The `_requireOnlyOperatorOrOwnerOf` function performs this verification by calling ERC721Upgradeable._isAuthorized with `msg.sender` as both the `owner` and `spender`:

```
function _requireOnlyOperatorOrOwnerOf(uint256 tokenId) internal view {
    // isApprovedOrOwner: caller has no rights on token
    require(ERC721Upgradeable._isAuthorized(msg.sender, msg.sender, tokenId),
↪  "FORBIDDEN");
}
```

Since `_isAuthorized` returns `true` when `owner == spender`, this check always evaluates to `true` regardless of the caller's identity:

```
function _isAuthorized(address owner, address spender, uint256 tokenId) internal
↪  view virtual returns (bool) {
```


SHERLOCK

```
    return
        spender != address(0) &&
        (owner == spender || isApprovedForAll(owner, spender) ||
↪   _getApproved(tokenId) == spender);
}
```

Any function implementing this check is vulnerable to bypass, but currently only `addToPosition` is affected. An attack scenario involves an attacker manipulating the `lockDuration` of another user's position by donating a small amount (1 wei). When users create a position, `lockDuration` dictates how long their asset is locked. This duration impacts rewards, voting eligibility, etc. If an attacker increases another user's position via `addToPosition`, the function recalculates lockDuration based on new `amountToAdd` and existing lock times:

```
// we calculate the avg lock time:
// lock_duration = (remainin_lock_time * staked_amount + amount_to_add *
↪  inital_lock_duration) / (staked_amount + amount_to_add)
uint256 remainingLockTime = _remainingLockTime(position);
uint256 avgDuration = (remainingLockTime * position.amount + amountToAdd *
↪  position.initialLockDuration)
    / (position.amount + amountToAdd);
```

```
function _remainingLockTime(StakingPosition memory position) internal view
↪  returns (uint256) {
    if ((position.startLockTime + position.lockDuration) <=
↪  _currentBlockTimestamp()) {
        return 0;
    }
    return (position.startLockTime + position.lockDuration) -
↪  _currentBlockTimestamp();
}
```

A malicious user can call `addToPosition` and donate 1 wei to a position, thereby manipulating the lockDuration of that position, which will have multiple consequences for the victim.

- Alice creates a position with 100e18 as `amount` and 2 weeks as `lockDuration`.
- 1 week later the Admin starts a new voting period.
- Given that Alice's position's `lockDuration` is equal to the `PeriodDuration`, which is 2 weeks, she is eligible to participate in voting.
- Bob (a malicious actor) calls the `addToPosition` and donate 1 wei to Alice's position
- Alice's `lockDuration` is recalculated, potentially reducing it to 1 week.

- Alice calls the `vote` but her transaction reverts due to `InsufficientLockTime` error.

  NOTE: There's no need to front-run Alice's transaction because whenever Bob performs this malicious `addToPosition`, the `lockDuration` of Alice's position will be manipulated. All that Bob needs to do is execute this action before Alice calls the vote, and it doesn't have to be immediately preceding Alice's transaction.

Coded PoC

Above scenario is implemented in this test Please make a file named `Ninja.t.sol` in this path: `/test/` and paste the following test code in it:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

import "forge-std/Test.sol";

import "../src/transparent/TransparentUpgradeableProxy2Step.sol";

import {ERC20Mock} from "./mocks/ERC20.sol";
import {MasterChefMock} from "./mocks/MasterChefMock.sol";
import {MlumStaking} from "../src/MlumStaking.sol";
import "../src/Voter.sol";
import {IVoter} from "../src/interfaces/IVoter.sol";
import "../src/interfaces/IBribeRewarder.sol";
import "../src/rewarders/BribeRewarder.sol";
import "../src/rewarders/RewarderFactory.sol";

contract Ninja is Test {
    address payable immutable DEV = payable(makeAddr("dev"));
    address payable immutable ALICE = payable(makeAddr("alice"));
    address payable immutable BOB = payable(makeAddr("bob"));

    address pool = makeAddr("pool");

    Voter private _voter;
    MlumStaking private _pool;

    ERC20Mock private _stakingToken;
    ERC20Mock private _rewardToken;
    ERC20Mock private _bribeRewardToken;

    BribeRewarder rewarder;
```


SHERLOCK

```
    RewarderFactory factory;

    function setUp() public {
        _bribeRewardToken = new ERC20Mock("Reward Token", "RT", 6);

        vm.prank(DEV);
        _stakingToken = new ERC20Mock("MagicLum", "MLUM", 18);

        vm.prank(DEV);
        _rewardToken = new ERC20Mock("USDT", "USDT", 6);

        vm.prank(DEV);
        address poolImpl = address(new MlumStaking(_stakingToken, _rewardToken));

        _pool = MlumStaking(
            address(
                new TransparentUpgradeableProxy2Step(
                    poolImpl, ProxyAdmin2Step(address(1)),
↪   abi.encodeWithSelector(MlumStaking.initialize.selector, DEV)
                )
            )
        );

        vm.prank(DEV);
        MasterChefMock mock = new MasterChefMock();

        address factoryImpl = address(new RewarderFactory());
        factory = RewarderFactory(
            address(
                new TransparentUpgradeableProxy2Step(
                    factoryImpl,
                    ProxyAdmin2Step(address(1)),
                    abi.encodeWithSelector(
                        RewarderFactory.initialize.selector, address(this), new
↪   uint8[](0), new address[](0)
                    )
                )
            )
        );
        address voterImpl = address(new Voter(mock, _pool,
↪   IRewarderFactory(address(factory))));

        _voter = Voter(
            address(
                new TransparentUpgradeableProxy2Step(
                    voterImpl, ProxyAdmin2Step(address(1)),
↪   abi.encodeWithSelector(Voter.initialize.selector, DEV)
```


```
                )
            )
        );

        factory.setRewarderImplementation(
            IRewarderFactory.RewarderType.BribeRewarder, IRewarder(address(new
↪   BribeRewarder(address(_voter))))
        );
        rewarder =
↪   BribeRewarder(payable(address(factory.createBribeRewarder(_bribeRewardToken,
↪   pool))));

        vm.prank(DEV);
        _voter.updateMinimumLockTime(2 weeks);
    }

    function test_MaliciousAddToPosition() public {
        IMlumStaking.StakingPosition memory position;

        _createPosition(ALICE, 100 ether, 2 weeks);
        assertEq(_pool.ownerOf(1), ALICE);

        _stakingToken.mint(BOB, 1 wei);

        position = _pool.getStakingPosition(1);
        assertEq(position.lockDuration, 2 weeks);
        assertEq(_remainingLockTime(position), 2 weeks);

        console.log("%s%s", "lockDuration Before Malicious addToPosition: ",
↪   position.lockDuration);

        skip(1 weeks);
        // 1 week later the Admin starts a voting period
        vm.prank(DEV);
        _voter.startNewVotingPeriod();
        assertEq(1, _voter.getCurrentVotingPeriod());

        // Alice is eligible to Vote because her position's lockDuration is = to
↪   the duration of the period
        // 2 weeks == 2 weeks
        assertEq(position.lockDuration, _voter.getPeriodDuration());

        // Bob maliciously adds 1 wei to Alice's position which will update her
↪   position and decreases her lockDuration
        vm.startPrank(BOB);
        _stakingToken.approve(address(_pool), 1 wei);
        _pool.addToPosition(1, 1 wei);
```


SHERLOCK

```
        vm.stopPrank();

        // Alice will lose her eligibility due to the lockDuration decrease that
↪   occurred after Bob maliciously added 1 wei to her position.
        // Now Alice's lockDuration is 1 week, which is less than the
↪   PeriodDuration which 2 weeks.
        position = _pool.getStakingPosition(1);
        assertLt(position.lockDuration, _voter.getPeriodDuration());
        assertEq(position.lockDuration, 1 weeks);

        console.log("%s%s", "lockDuration After Malicious addToPosition:  ",
↪   position.lockDuration);

        // Alice calls the vote function but her tx reverts due to the
↪   InsufficientLockTime error
        vm.prank(ALICE);
        vm.expectRevert(
            abi.encodeWithSelector(
                IVoter.IVoter__InsufficientLockTime.selector
            )
        );
        _voter.vote(1, _getDummyPools(), _getDeltaAmounts());
    }

    /*
    ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
    |||||||||||||||||||||||| INTERNAL FUNCTIONS ||||||||||||||||||||||||||||||||
    ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
    */

    function _createPosition(address user, uint256 amount, uint256 duration)
↪   internal {
        _stakingToken.mint(user, 100 ether);

        vm.startPrank(user);
        _stakingToken.approve(address(_pool), amount);
        _pool.createPosition(amount, duration);
        vm.stopPrank();
    }

    function _getDeltaAmounts() internal pure returns (uint256[] memory
↪   deltaAmounts) {
        deltaAmounts = new uint256[](1);
        deltaAmounts[0] = 1e18;
    }

    function _getDummyPools() internal view returns (address[] memory pools) {
```


```
        pools = new address[](1);
        pools[0] = pool;
    }

    function _remainingLockTime(IMlumStaking.StakingPosition memory position)
↪   internal view returns (uint256) {
        if ((position.startLockTime + position.lockDuration) <=
↪   _currentBlockTimestamp()) {
            return 0;
        }
        return (position.startLockTime + position.lockDuration) -
↪   _currentBlockTimestamp();
    }

    function _currentBlockTimestamp() internal view virtual returns (uint256) {
        return block.timestamp;
    }
}
```

Run the test:

```
forge test --mt test_MaliciousAddToPosition -vv
```

Output:

```
Compiler run successful!

Ran 1 test for test/Ninja.t.sol:Ninja
[PASS] test_MaliciousAddToPosition() (gas: 624706)
Logs:
  lockDuration Before Malicious addToPosition: 1209600
  lockDuration After Malicious addToPosition:  604800

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.64ms (340.63s CPU
↪  time)
```

Impact

This bug has multiple impacts on rewards calculation, voting eligibility, etc., as highlighted in this report, particularly preventing users from participating in voting, thereby denying them access to rewards.

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/MlumStaking.sol#L398 https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/MlumStaking.sol#L142 https://github.com/sherlock-audit/2024-06-magicsea/blob/main/magicsea-staking/src/MlumStaking.sol#L409

Tool used

Manual Review

Recommendation

```
    function _requireOnlyOperatorOrOwnerOf(uint256 tokenId) internal view {
        // isApprovedOrOwner: caller has no rights on token
-       require(ERC721Upgradeable._isAuthorized(msg.sender, msg.sender,
↪   tokenId), "FORBIDDEN");
+       require(ERC721Upgradeable._isAuthorized(ownerOf(tokenId), msg.sender,
↪   tokenId), "FORBIDDEN");
    }
```

Discussion

0xSmartContract

Users Can Add Amounts Instead of Each Other: We see that the `_requireOnlyOperatorOrOwnerOf` function always returns true using the msg.sender and msg.sender arguments, and therefore any user can add amounts to other users' staking positions.

This can lead to harm to the positions of others and lead to involuntary changes.

To solve this problem, the `_requireOnlyOperatorOrOwnerOf` function must be updated and it must be properly checked whether the caller is authorized or not.

0xHans1

PR: https://github.com/metropolis-exchange/magicsea-staking/pull/9

We fixed this issue already for an other audit. Basically we removed all isApprover... / requireOnlyOperator... checks and check for ownership instead. The linked PR just highlight the _checkOwnerOf function.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/metropolis-exchange/magicsea-staking/pull/9

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-8: Unclaimed rewards when emergency withdrawing are not redistributed in `MasterChef` and `MlumStaking`

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/460

Found by

0xAnmol, KupiaSec, iamnmt, pashap9990

Summary

Unclaimed rewards when emergency withdrawing are not redistributed in `MasterChef` and `MlumStaking`.

Vulnerability Detail

Users can call `MasterChef#emergencyWithdraw`, `MlumStaking#emergencyWithdraw` to withdraw tokens without claiming the rewards.

But the unclaimed rewards are not redistributed in `emergencyWithdraw` call, which will left the rewards stuck in the contracts and lost forever. Other users can not claim the rewards and the protocol can not redistribute the rewards.

Impact

Unclaimed rewards when emergency withdrawing in `MasterChef` and `MlumStaking` will stuck in the contracts and lost forever.

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/MasterchefV2.sol#L326

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/MlumStaking.sol#L536

Tool used

Manual Review

Recommendation

Redistribute the unclaimed rewards in `emergencyWithdraw`

`MasterchefV2.sol`

SHERLOCK

```
    function emergencyWithdraw(uint256 pid) external override {
        Farm storage farm = _farms[pid];

        uint256 balance = farm.amounts.getAmountOf(msg.sender);
        int256 deltaAmount = -balance.toInt256();

-       farm.amounts.update(msg.sender, deltaAmount);

+       (uint256 oldBalance, uint256 newBalance, uint256 oldTotalSupply,) =
 ↪  farm.amounts.update(msg.sender, deltaAmount);

+       uint256 totalLumRewardForPid = _getRewardForPid(farm.rewarder, pid,
 ↪  oldTotalSupply);
+       uint256 lumRewardForPid = _mintLum(totalLumRewardForPid);

+       uint256 lumReward = farm.rewarder.update(msg.sender, oldBalance,
 ↪  newBalance, oldTotalSupply, lumRewardForPid);
+       lumReward = lumReward + unclaimedRewards[pid][msg.sender];
+       unclaimedRewards[pid][msg.sender] = 0;

+       farm.rewarder.updateAccDebtPerShare(oldTotalSupply, lumReward);

        farm.token.safeTransfer(msg.sender, balance);

        emit PositionModified(pid, msg.sender, deltaAmount, 0);
    }
```

MlumStaking.sol

```
function emergencyWithdraw(uint256 tokenId) external override nonReentrant {
        _requireOnlyOwnerOf(tokenId);

        StakingPosition storage position = _stakingPositions[tokenId];

        // position should be unlocked
        require(
            _unlockOperators.contains(msg.sender)
                || (position.startLockTime + position.lockDuration) <=
 ↪  _currentBlockTimestamp() || isUnlocked(),
            "locked"
        );
        // emergencyWithdraw: locked

+       _updatePool();
```


SHERLOCK

```
+       uint256 pending = position.amountWithMultiplier * _accRewardsPerShare /
 ↪  PRECISION_FACTOR - position.rewardDebt;

+       _lastRewardBalance = _lastRewardBalance - pending;

        uint256 amount = position.amount;

        // update total lp supply
        _stakedSupply = _stakedSupply - amount;
        _stakedSupplyWithMultiplier = _stakedSupplyWithMultiplier -
 ↪  position.amountWithMultiplier;

        // destroy position (ignore boost points)
        _destroyPosition(tokenId);

        emit EmergencyWithdraw(tokenId, amount);
        stakedToken.safeTransfer(msg.sender, amount);
    }
```

Discussion

0xSmartContract

This report was chosen as the main report because it states the findings `MasterChef#emergencyWithdraw` and `MlumStaking#emergencyWithdraw` in a single report.

0xSmartContract

When emergency withdrawing are made, Unclaimed rewards in `MasterChef` and `MlumStaking` contracts remain stuck in the contracts and are lost forever

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/metropolis-exchange/magicsea-staking/pull/34

chinmay-farkya

Escalate

Some issues have been incorrectly included in the duplicates here

#536 says “Loss of reward tokens during emergency withdrawing” but thats the whole point of emergency Withdraw and this has been clearly mentioned in the comments. The user is willingly letting go off rewards during an emergency acc to expected design. This is not a bug but a feature. #671 is not a duplicate and is invalid due to the same reason above. Both these do not talk about redistribution or

stuck rewards, but only about the withdrawer himself losing out on rewards, which is expected behaviour as per comments.

Apart from these, the remaining issues are actually two sets of issues clubbed together incorrectly.

Set 1 is about “non-redistribution of rewards of stakers who use emergencyWithdraw for exit”. The root cause is that the accRewardPerShare or lastRewardBalance are not properly adjusted so the rewards that were let go off by the staker in MasterChef and MLUMStaking are left stuck. And the fix is to adjust accRewardPerShare and lastRewardBalance accordingly in emergencyWithdraw logic or add a sweep function. Here, briber cant get back the rewards that will get stuck.

Set 1 : #460, #589, #428. note that this also has another duplicate not mentioned here ie. #368

Set 2 is about the rewards lost in extraRewarder attached to a farm, if the extraRewarder is unlinked from the farm while it still stores unclaimed rewards of various stakers of that farm. Root cause is the non-existence of a direct claiming method in MasterChefRewarder contract and the fix is to add such a method. Here, the stakers will lose out on rewards they had “already earned” in the form of extra rewards. This is unrelated to the funds stuck because masterchefrewarder inherits from baserewarder which has a sweep function, but the problem here is about the deserved users unable to claim their earned rewards after unlinking.

We can clearly see that these two sets are completely different according to root cause, fix ,and the code segment & actor affected.

Set 2 : #342, #468, #559, #611, #649, #40

#559 is invalid because the issue description is wrong. It says “funds will be stuck forever”. But the truth is the sweep function will be used to retrieve the funds as _getTotalSupply will return 0 after becoming unlinked. So the funds are not stuck, the non-existence of a direct claim method is the problem as described above. In short, the description and impact of this submission is completely wrong, hence this is invalid.

#611 is invalid because the issue description is completely wrong. As we can see in the code, if getTotalSupply returns zero, then the whole of contract’s balance will be sweeped out. And the getTotalsupply returns zero after becoming unlinked. So the rewards are not permanently stuck and can be sweeped out by owner. Instead, the problem is something different not identified by this issue. Hence, this is invalid due to wrong description and impact.

#40 is invalid as it talks about something else around the rewards calculations using timestamp which is not even relevant to the context of this discussion. The Masterchef’s lastUpdateTimestamp is not relevant to masterchefRewarder

contract’s distribution of "extra rewards".

So Set 2 has only 3 valid issues : #342, #468, #649

Both these sets are two separate valid Medium findings

sherlock-admin3

> Escalate
>
> Some issues have been incorrectly included in the duplicates here
>
> #536 says “Loss of reward tokens during emergency withdrawing” but thats the whole point of emergency Withdraw and this has been clearly mentioned in the comments. The user is willingly letting go off rewards during an emergency acc to expected design. This is not a bug but a feature. #671 is not a duplicate and is invalid due to the same reason above. Both these do not talk about redistribution or stuck rewards, but only about the withdrawer himself losing out on rewards, which is expected behaviour as per comments.
>
> Apart from these, the remaining issues are actually two sets of issues clubbed together incorrectly.
>
> Set 1 is about “non-redistribution of rewards of stakers who use emergencyWithdraw for exit”. The root cause is that the accRewardPerShare or lastRewardBalance are not properly adjusted so the rewards that were let go off by the staker in MasterChef and MLUMStaking are left stuck. And the fix is to adjust accRewardPerShare and lastRewardBalance accordingly in emergencyWithdraw logic or add a sweep function. Here, briber cant get back the rewards that will get stuck.
>
> Set 1 : #460, #589, #428. note that this also has another duplicate not mentioned here ie. #368
>
> Set 2 is about the rewards lost in extraRewarder attached to a farm, if the extraRewarder is unlinked from the farm while it still stores unclaimed rewards of various stakers of that farm. Root cause is the non-existence of a direct claiming method in MasterChefRewarder contract and the fix is to add such a method. Here, the stakers will lose out on rewards they had “already earned” in the form of extra rewards. This is unrelated to the funds stuck because masterchefrewarder inherits from baserewarder which has a sweep function, but the problem here is about the deserved users unable to claim their earned rewards after unlinking.
>
> We can clearly see that these two sets are completely different according to root cause, fix ,and the code segment & actor affected.
>
> Set 2 : #342, #468, #559, #611, #649, #40

#559 is invalid because the issue description is wrong. It says “funds will be stuck forever”. But the truth is the sweep function will be used to retrieve the funds as _getTotalSupply will return 0 after becoming unlinked. So the funds are not stuck, the non-existence of a direct claim method is the problem as described above. In short, the description and impact of this submission is completely wrong, hence this is invalid.

#611 is invalid because the issue description is completely wrong. As we can see in the code, if getTotalSupply returns zero, then the whole of contract’s balance will be sweeped out. And the getTotalsupply returns zero after becoming unlinked. So the rewards are not permanently stuck and can be sweeped out by owner. Instead, the problem is something different not identified by this issue. Hence, this is invalid due to wrong description and impact.

#40 is invalid as it talks about something else around the rewards calculations using timestamp which is not even relevant to the context of this discussion. The Masterchef’s lastUpdateTimestamp is not relevant to masterchefRewarder contract’s distribution of "extra rewards".

So Set 2 has only 3 valid issues : #342, #468, #649

Both these sets are two separate valid Medium findings

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

blckhv

This should be low, code comments are the main source of truth when readme contains no information:

```
* @dev Emergency withdraws tokens from a farm, without claiming any rewards.
```

Additionally, this is an opportunity loss, since user who has called `emergencyWithdraw` can after that call `claim` and this will set the needed params: https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/MasterchefV2.sol#L551-L554

```
function _modify(uint256 pid, address account, int256 deltaAmount, bool
↪  isPayOutReward) private {
        Farm storage farm = _farms[pid];
        Rewarder.Parameter storage rewarder = farm.rewarder;
        IMasterChefRewarder extraRewarder = farm.extraRewarder;
```

```
        (uint256 oldBalance, uint256 newBalance, uint256 oldTotalSupply,) =
↪   farm.amounts.update(account, deltaAmount);

        uint256 totalLumRewardForPid = _getRewardForPid(rewarder, pid,
↪   oldTotalSupply);
        uint256 lumRewardForPid = _mintLum(totalLumRewardForPid);

        uint256 lumReward = rewarder.update(account, oldBalance, newBalance,
↪   oldTotalSupply, lumRewardForPid);

        if (isPayOutReward) {
            lumReward = lumReward + unclaimedRewards[pid][account];
            unclaimedRewards[pid][account] = 0;
            if (lumReward > 0) _lum.safeTransfer(account, lumReward);
        }
```

So no locked funds.

iamnmt

@blckhv

> since user who has called `emergencyWithdraw` can after that call claim and this will set the needed params So no locked funds.

I believe this claim is not correct. In the `emergencyWithdraw` function, the unclaimed rewards are not added to `unclaimedRewards[pid][account]`. Because of that, funds from the last time a user, who is emergency withdrawing, called `_modify` to the time the user called `emergencyWithdraw` are still locked.

MatinR1

Regarding the comment on issue #536, I would like to provide further clarification. While it is true that the emergency withdraw function is designed to allow users to withdraw their staked tokens immediately, forfeiting any accrued rewards, my report highlights a different concern. The issue is not about the intentional forfeiture of rewards during an emergency withdrawal, but rather the loss of unclaimed rewards due to the contract's logic.

I explicitly emphasized the **unclaimed** rewards in the proof of concept (POC) part. Throughout the entire report, my focus was on the unclaimed rewards. I understand that forfeiting rewards during an emergency withdrawal is an intended feature, as stated in the function's NatSpec description. If my mitigation suggestion was off-target, it was because my primary concern was ensuring that unclaimed rewards were properly addressed.

Therefore, I maintain that this issue should be considered a bug, as it impacts the overall integrity and expected functionality of the rewards distribution process.

0xSmartContract

Escalate 1 : 536 incorrectly included in the duplicates here. Answer. 1 : I disagree

Escalate 1 Issue number 460: In this issue, it is argued that the loss of accumulated rewards during the emergency withdrawal process is a problem and that these rewards should be redistributed to other users or the system.

Issue number 536: The same problem is detected in this issue, but the solution proposal is slightly different. Here, it is suggested that the accumulated rewards during the emergency withdrawal process should be given to the user.

If we break the basic problem and navigate inside, we will probably go to very different places, do not break away from the main picture.

Escalate 2 Escalate 2 : 671 incorrectly included in the duplicates here. Answer. 2 : I disagree . (Same reasons as above)

Both issues involve losing rewards during the emergency withdrawal process. Issue 460 focuses on the failure to redistribute rewards, while issue 671 focuses on the user losing their rewards. However, in both cases, the fundamental problem is that the emergency withdrawal process does not handle rewards correctly.

Both issues affect the integrity of the system. Either the failure to redistribute rewards (460) or the user losing their rewards (671) prevents the system from operating fairly and efficiently. These are two aspects of the same problem.

A similar approach is required to solve both problems: correctly calculating and processing rewards during the emergency withdrawal process. This could be either giving the rewards to the user or redistributing them fairly throughout the system.

The code changes that would be made to solve both problems would likely be similar. For example, calling the `_modify` function or updating the reward calculations.

Escalate 3 Escalate 3 : Set 1 and Set 2 Answer 3 : I disagree .

All these issues (Set 1 and Set 2) are caused by deficiencies in the reward distribution mechanism. Whether it is in the emergency exit case or in the case of extraRewarder disconnection, the main problem is that users and the project cannot receive the rewards.

The solutions suggested for both sets (fixing accRewardPerShare and lastRewardBalance, adding a sweep function, adding a direct request method) actually serve the same basic purpose: providing access to the rewards.

Important Note: Should the project receive these rewards or should the users receive them? This part is an architectural decision and the project can decide on this, both can happen together (You probably want it to be separated into two parts because of this dilemma anyway)

The code changes to be made to solve these problems require a single holistic approach.

As a result, instead of evaluating these problems as separate sets, it would be better to consider them all as different aspects of the same basic problem.

---

chinmay-farkya

@0xSmartContract for "Escalate 3" consider that both set of problems are in different contracts altogether.

While the set 1 is about MLUMStaking and MasterChef, the set 2 is about masterchefrewarder.

Even if these two sets are decided to remain clubbed as one, I will provide additional arguments to prove that the other issues I listed as invalid are actually invalid.

Regarding "Escalate 1" : quoted from #536 : "The absence of the harvesting mechanism in emergencyWithdraw() can lead to significant loss of accumulated rewards for users who invoke this function." which clearly is implying to the loss caused to the withdrawer themselves, which is expected behavior. These two issues #536 and #671 are saying that the withdrawer of the position himself is losing rewards because of logic not calling _harvestPosition (see the recommendation mentioned in that issue)

Same goes for "Escalate 2"

These two and the other ones I mentioned as invalid in original escalation Set 2 are invalid and not part of either Set 1, nor of Set 2 and not a duplicate even if these two sets remain combined into a single issue (please go over the reasoning in original escalation for why #559, #611, #40 are invalid issues wrongly validated.)

0xSmartContract

If we break the basic issue and navigate inside, we will probably go to very different places (differently contracts , differently functions, differently recommendation, differently others), do not break away from the main picture.

yash-0025

My Issue #660 also shows that the rewards fund is stucked for the user during emergency withdraw with the POC screenshot in the 2nd scenario I provided and the same issue which depicts the same Issue is this #460 and it is considered valid.

So isn't it a mistaken duped rather than an invalid ? And the issue #660 is escalated and the reason given is invalid.

0xSmartContract

> @0xSmartContract for "Escalate 3" consider that both set of problems are in different contracts altogether.
>
> While the set 1 is about MLUMStaking and MasterChef, the set 2 is about masterchefrewarder.
>
> Even if these two sets are decided to remain clubbed as one, I will provide additional arguments to prove that the other issues I listed as invalid are actually invalid.
>
> Regarding "Escalate 1" : quoted from #536 : "The absence of the harvesting mechanism in emergencyWithdraw() can lead to significant loss of accumulated rewards for users who invoke this function." which clearly is implying to the loss caused to the withdrawer themselves, which is expected behavior. These two issues #536 and #671 are saying that the withdrawer of the position himself is losing rewards because of logic not calling _harvestPosition (see the recommendation mentioned in that issue)
>
> Same goes for "Escalate 2"
>
> These two and the other ones I mentioned as invalid in original escalation Set 2 are invalid and not part of either Set 1, nor of Set 2 and not a duplicate even if these two sets remain combined into a single issue (please go over the reasoning in original escalation for why #559, #611, #40 are invalid issues wrongly validated.)

1 - If we agree that the two sets remain as one, we can move on to the invalid ones

2- When I re-evaluate within the framework of your views, 611, 559 and 40 and duplicates can be considered invalid When `getTotalSupply` is zero and the contract is unlinked, the entire balance can be sweeped. This indicates that `unclaimed rewards` are not actually permanently frozen and can be sweeped by the owner. Therefore, the description and effect of issue #611 #559 and #40 are incorrect and invalid.

chinmay-farkya

> All these issues (Set 1 and Set 2) are caused by deficiencies in the reward distribution mechanism. Whether it is in the emergency exit case or in the case of extraRewarder disconnection, the main problem is that users and the project cannot receive the rewards. The solutions suggested for both sets (fixing accRewardPerShare and lastRewardBalance, adding a

> sweep function, adding a direct request method) actually serve the same basic purpose: providing access to the rewards.

That is not correct. When they can't be solved with a single fix at a single place in code, and do not stem from the same root cause, how can they be duped ? From my understanding, dups are based on the same issues, and not on the "same contract" or "same code component/mechanism"

Moreover, the statement "The solutions suggested for both sets (fixing accRewardPerShare and lastRewardBalance, adding a sweep function, adding a direct request method) actually serve the same basic purpose: providing access to the rewards." is actually incorrect.

As i pointed out in my original escalation, set 1 is about missing re-distribution of rewards : stakers willingly forefeited their rewards by using emergencyWithdraw and these rewards now "do not belong to anyone" and thus they will get stuck because of lcak of redistribution.

Set 2 is about missing way to claim rewards : stakers had earned rewards which they are "not willing to forfeit" but when extrarewarder is unlinked, they are forcefully forefeited for everyone. These rewards actually belong to these stakers as they had already accrued them in storage. (note that I'm not talking about the rewards that had not accrued). Now this is not the same as funds get stuck, because these should not be sweeped by the protocol, "they belong to the stakers" and require a direct claim method.

I think that these are clearly not dups.

0xSmartContract

I think we have made progress on the issue To summarize the #460 issue;

@chinmay-farkya's suggestion; Set 1 : #460, #589, #428, #368(+368's duplicates) Set 2 : #342, #468, #649, Invalids : #536 , #671 ,#559, #611,#40

Lead Judge suggestion ; Only one set: #460, #589, #428,#342, #468, #649 , #536, #671 Invalids : #559, #611,#40

As the Lead Judge, I appreciate your detailed analysis, but I believe there are strong reasons to consider these issues as part of a single set. Here's why:

Overarching Problem: While the specific mechanisms differ, all these issues fundamentally relate to the mishandling of rewards in exceptional circumstances (emergency withdrawals or unlinking of extra rewarders). The core problem in each case is that rewards are not properly accounted for or distributed.

System Impact: Both scenarios (emergency withdrawals and unlinking extra rewarders) result in a similar outcome - users potentially losing access to rewards they've accrued. This represents a systemic flaw in the reward distribution mechanism of the protocol.

Holistic Solution Approach: While the exact code fixes may differ, addressing these issues requires a comprehensive review and redesign of how the protocol handles rewards in edge cases. This suggests a unified approach to solving the problem, rather than treating them as entirely separate issues.

While I acknowledge that the specific mechanisms and code locations differ, I believe the similarities in the fundamental problem, impact, and required solution approach justify grouping these issues together. This holistic view allows for a more comprehensive understanding and resolution of the protocol's reward distribution challenges.

Regarding the invalid issues (#559, #611, #40), I agree with your assessment. These issues either mischaracterize the problem or focus on aspects that are not directly related to the core issue of reward mishandling in exceptional circumstances.

yash-0025

> My Issue #660 also shows that the rewards fund is stucked for the user during emergency withdraw with the POC screenshot in the 2nd scenario I provided and the same issue which depicts the same Issue is this #460 and it is considered valid. So isn't it a mistaken duped rather than an invalid ? And the issue #660 is escalated and the reason given is invalid.

@0xSmartContract May I know about this issue!!!

WangSecurity

I'll go in the same order that the escalation comment has:

1. #536 -- I agree, the issue basically says if Bob calls `emergencyWithdraw` they lose their rewards. It's invalid, it's intended.
2. #671 -- the same, invalid and intended.
3. I agree that there are 2 sets of issues with different root causes. This report is about tokens being stuck and undistributed after someone uses `emergencyWithdraw`. #342 (and its duplicates) are about losing rewards when the Rewarder is unlinked.

But, I need a clarification about the rewarder, is it set by the MagicSea team or anyone can set it as a Bribe rewarder for example?

#660 is the same as #536 and #671, it just explains how the user loses their rewards during the emergency withdraw which is the exact purpose of this function. It does mention the loss for the protocol in the impact, but no explanation of how the protocol loses here. Hence, remains invalid.

Also, excuse me for the delay here.

chinmay-farkya

> But, I need a clarification about the rewarder, is it set by the MagicSea team or anyone can set it as a Bribe rewarder for example?

It is set by the magicsea team I guess. But once they do unlink it, the impact is that the way of claiming users' rewards is immediately cut off from the extrarewarder. The admin does not have a workaround to time this action of "unlinking". This admin action harms everybody.

MatinR1

dear @WangSecurity, I would kindly like to request a re-evaluation of my bug report #536 . My primary concern was focused on the unclaimed rewards, which I clearly emphasized in my report. The issue revolves around how these unclaimed rewards are handled within the system. I believe there might be a misunderstanding regarding the context of my findings.

Thank you for your consideration.

WangSecurity

> It is set by the magicsea team I guess. But once they do unlink it, the impact is that the way of claiming users' rewards is immediately cut off from the extrarewarder. The admin does not have a workaround to time this action of "unlinking". This admin action harms everybody

Thank you for this clarification, but I'm afraid the following rule should apply:

> An admin action can break certain assumptions about the functioning of the code. Example: Pausing a collateral causes some users to be unfairly liquidated or any other action causing loss of funds. This is not considered a valid issue.

In this case, the admin unlinking the rewarder leads to users losing these rewards completely. Hence, I believe the rule is applicable and the second set should be invalid.

> I would kindly like to request a re-evaluation of my bug report https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/536 . My primary concern was focused on the unclaimed rewards, which I clearly emphasized in my report. The issue revolves around how these unclaimed rewards are handled within the system. I believe there might be a misunderstanding regarding the context of my findings.

I still stand by my words in the previous comment. #536 is only a recommendation to allow users who use `emergencyWithdraw` to allow them get the rewards later.

Hence, I decide to accept the escalation, de-dup #536, #671, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/342, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/468, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/559,

https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/611, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/649, https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/40

iamnmt

@WangSecurity

To summarize the second set of issues:

Internal pre-conditions:

1. An extra rewarder is set in `MasterchefV2` for a pool.
2. Any users have unclaimed extra rewards in the extra rewarder.

Vulnerability path: Admin calls to `MasterchefV2#setExtraRewarder(pid, extraRewarder)` with **ANY** `extraRewarder`, then the current `extraRewarder` will be unlinked. Another way to phrase this vulnerability path is the admin triggers the 'MasterChefRewarder#unlink function.

I believe that it is intended for the admin calls to `MasterchefV2#setExtraRewarder(pid, extraRewarder)` even when the first internal pre-condition is met, in other words, the function `MasterchefV2#setExtraRewarder` is intended for **changing** the extra rewarder, because of these reasons:

1. In `MasterchefV2#setExtraRewarder` there is no check for the current extra rewarder to be `address(0)`.
2. There is a `unlink` function implemented in the extra rewarder, meaning it is expected for the extra rewarder to be unlinked.

I believe the second set of issues is valid because the vulnerability path is fulfilled the note of the fifth item of the third section in the judging rules:

> 5. (External) Admin trust assumptions: ... ... Note: if the attack path is possible with any possible value, it will be a valid issue.

Also, may I ask what is the "certain assumptions about the functioning of the code" in the context of this issue?

chinmay-farkya

> An admin action can break certain assumptions about the functioning of the code. Example: Pausing a collateral causes some users to be unfairly liquidated or any other action causing loss of funds. This is not considered a valid issue.

This statement is only true imo when the admin has a workaround alternate way of doing things : such that it would not lead to losses, and so they are trusted to instead go for this alternate way.

Here, in the set 2 of issues, there is no alternate way for the admin. Setting a new extrarewarder will definitely cause loss of rewards for "all users" every time the admin uses this function, all unclaimed rewards in the previous extrarewarder contract will not be able to be claimed by users who earned them.

WangSecurity

> I believe the second set of issues is valid because the vulnerability path is fulfilled the note of the fifth item of the third section in the judging rules:

It's correct, but I believe this rule is not applicable here. What I'm applying is the fifth item in the section called
"List of Issue categories that are not considered valid".

> Also, may I ask what is the "certain assumptions about the functioning of the code" in the context of this issue?

In this case, the assumption is that the voters should be able to get their extra rewards even if the extraRewarder is unlinked. In fact, the admin unlinking extraRewarder leads to users losing their extra rewards.

> This statement is only true imo when the admin has a workaround alternate way of doing things : such that it would not lead to losses, and so they are trusted to instead go for this alternate way

Fair point, but I believe the rule doesn't imply anything like this. The rule is about the admin's actions causing loss of funds, e.g. pausing collateral leads to unfair liquidations (users cannot add or repay collateral) or the users losing their extra rewards if the admin unlinks the contract with extra rewards.

Hence, the decision remains as expressed in this comment. The escalation will be accepted.

chinmay-farkya

So is the verdict that "admin actions that do not have an alternate workaround way and will most definitely lead to loss of funds for users : is allowed to happen and will not be treated as a bug" ? That sounds illogical imo.

I will leave the decision to you, and ask for changing this rule in the future.

iamnmt

@WangSecurity

Why is the rule "List of Issue categories that are not considered valid" applied but not this rule?

> **(External) Admin trust assumptions:** When a function is access restricted, only values for specific function variables mentioned in the README can be taken into account when identifying an attack path. If no

> values are provided, the (external) admin is trusted to use values that will not cause any issues. Note: if the attack path is possible with any possible value, it will be a valid issue. Exception: It will be a valid issue if the attack path is based on a value currently live on the network to which the protocol will be deployed.

Invalidating the second set of issues is assuming that the admin will never trigger the `MasterChefRewarder#unlink` function, even though the function is implemented in the codebase. Shouldn't this be "breaking core contract functionality"?

This is my final comment on this issue. Hoping that the contest's rules will be updated to reflect admin related issues better.

WangSecurity

> Why is the rule "List of Issue categories that are not considered valid" applied but not this rule?

Because this issue is not connected to admin set variables. Here we are talking about the admin's actions, not about values that lead to an issue.

> Invalidating the second set of issues is assuming that the admin will never trigger the MasterChefRewarder#unlink function, even though the function is implemented in the codebase. Shouldn't this be "breaking core contract functionality"?

I didn't say that I invalidate this issue because "the admin will never trigger the `unlink` function". I'm invalidating this issue because admin unlinking the extra rewarder leads to a loss of unclaimed rewards. I agree that the admin will do that and call this function, but it's still the admin action that leads to a loss of funds. Excuse me if my previous answer wasn't precise enough, if you still don't understand my reasoning, please ask questions.

The decision remains the same as expressed here.

chinmay-farkya

> I agree that the admin will do that and call this function, but it's still the admin action that leads to a loss of funds

Then the rule has to be relooked imo to clarify such situations.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- chinmay-farkya: accepted

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-9: Lack of support for fee on transfer, rebasing and tokens with balance modifications outside of transfers.

Source: https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545

The protocol has acknowledged this issue.

Found by

0uts1der, 0xAnmol, 0xNilesh, 0xWhitehat, 4rdiii, AuditorPraise, Bauchibred, ChinmayF, Ericselvig, Honour, Hunter, John_Femi, KiroBrejka, KupiaSec, LeFy, MSaptarshi, NoOne, PASCAL, PUSH0, Reentrants, Ryonen, Silvermist, Smacaud, StraawHaat, TessKimy, Tri-pathi, Vancelot, WildSniper, ZanyBonzy, bbl4de, blockchain555, dany.armstrong90, dhank, dimulski, dod4ufn, gajiknownnothing, hunter_w3b, iamnmt, jennifer37, karsar, kmXAdam, minhquanym, neogranicen, novaman33, pinalikefruit, radin200, rbserver, scammed, sh0velware, sheep, slowfi, tedox, typicalHuman, walter, web3pwn, yixxas, zarkk01

Summary

The protocol wants to work with various ERC20 tokens, but in certain cases doesn't provide the needed support for tokens that charge a fee on transfer, tokens that rebase (negatively/positively) and overall, tokens with balance modifications outside of transfers.

Vulnerability Detail

The protocol wants to work with various ERC20 tokens, but still handles various transfers without querying the amount transferred and amount received, which can lead a host of accounting issues and the likes downstream.

For instance, In MasterchefV2.sol during withdrawals or particularly emergency withdrawals, the last user to withdraw all tokens will face issues as the amount registered to his name might be signifiacntly lesser than the token balance in the contract, which as a result will cause the withdrawal functions to fail. Or the protocol risks having to send extra funds from their pocket to coverup for these extra losses due to the fees. This is because on deposit, the amount entered is deposited as is, without accounting for potential fees.

```
function deposit(uint256 pid, uint256 amount) external override {
    _modify(pid, msg.sender, amount.toInt256(), false);
```

```
    if (amount > 0) _farms[pid].token.safeTransferFrom(msg.sender,
↪   address(this), amount);
}
```

Some tokens like stETH have a 1 wei corner case in which during transfers the amount that actually gets sent is actually a bit less than what has been specified in the transaction.

Impact

On a QA severity level, tokens received by users will be less than emitted to them in the event. On medium severity level, accounting issues, potential inability of last users to withdraw, potential loss of funds from tokens with airdrops, etc.

Code Snippet

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/MasterchefV2.sol#L287

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/MasterchefV2.sol#L298

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/MasterchefV2.sol#L309

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/MasterchefV2.sol#L334

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/MlumStaking.sol#L559

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/MlumStaking.sol#L649

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/MlumStaking.sol#L744

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/MlumStaking.sol#L747

https://github.com/sherlock-audit/2024-06-magicsea/blob/42e799446595c542eff9519353d3becc50cdba63/magicsea-staking/src/rewarders/BribeRewarder.sol#L120

Tool used

Manual Code Review

Recommendation

Recommend implementing a measure like that of the `_transferSupportingFeeOnTransfer` function that can correctly handle these transfers. A sweep function can also be created to help with positive rebases and airdrops.

Discussion

0xSmartContract

Upon thorough review of the findings categorized under the "Weird Token Issue" umbrella, it is evident that these issues root from the non-standard behaviors exhibited by certain ERC20 tokens.

These tokens, referred to as "weird tokens," include fee-on-transfer tokens, rebasing tokens (both positive and negative), tokens with low or high decimals, and tokens with other non-standard characteristics.

- The ERC20 standard is known for its flexibility and minimal semantic requirements. This flexibility, while useful, leads to diverse token implementations that deviate from the standard behaviors expected by smart contracts. The issues reported are a direct consequence of this looseness in the ERC20 specification.
- Fee-on-transfer tokens deduct a portion of the transferred amount as a fee, leading to discrepancies between the expected and actual transferred amounts. This impacts functions like `deposit()`, `withdraw()`, and reward calculations.
- Rebasing tokens adjust balances periodically, either increasing (positive rebase) or decreasing (negative rebase) token balances. This behavior causes issues with static balance tracking and can lead to inaccurate accounting and loss of funds.

Detail of Issues List and Analysis
https://gist.github.com/0xSmartContract/4e7e9fc500f7d6e6061d94ffde6c2206

Given README explicitly allowing for weird tokens, it makes sense to group all related findings under the umbrella of "Weird ERC20 Tokens".

Here are the technical and practical reasons why this grouping is justified:

According to Sherlock rules, the "Weird Token" List is unique and all of these issues are in this single list and are not separated.

https://docs.sherlock.xyz/audits/judging/judging#vii.-list-of-issue-categories-that-are-not-considered-valid

> Non-Standard tokens: Issues related to tokens with non-standard behaviors, such as [weird-tokens] (https://github.com/d-xo/weird-erc20) are not considered valid by default unless these tokens are explicitly mentioned in the README.

Unified Theme and Clarity:By categorizing these issues together, the report maintains a coherent theme, making it easier for auditers and Sponsor to understand the underlying problem of dealing with unconventional ERC20 tokens.

Specification Looseness: The ERC20 specification is known to be loosely defined, leading to diverse implementations. Many tokens deviate from standard behaviors, causing unexpected issues when integrated into smart contracts.

Common Problems: Issues like fee-on-transfer, rebasing (both positive and negative), low/high decimals, and missing return values are common among these tokens, causing similar types of problems across different functions and contracts.

Here are key issues grouped under the "Weird ERC20 Tokens" category:

- `Fee on Transfer`: Tokens deduct a fee during transfers, leading to discrepancies between the expected and actual transferred amounts.
- `Rebase (Negative and Positive)`: Tokens adjust balances periodically, causing issues with static balance tracking in smart contracts.
- `Low/High Decimals`: Tokens with unusual decimal places lead to precision loss and overflow issues.
- `Missing/False Return Values`: Tokens that do not follow standard return values on transfers or approvals, causing unexpected failures.
- `Special Behaviors`: Tokens like cUSDCv3 that have unique handling for specific amounts, leading to manipulation opportunities.

novaman33

Escalate, This grouping is too general. Issues have different root causes and also different severity.

sherlock-admin3

> Escalate, This grouping is too general. Issues have different root causes and also different severity.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xSmartContract

I guess you didn't see this comment before Escalade;

https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545#issuecomment-2251178050

Grouping the category in this way allows auditors and sponsors to better understand the general problems caused by non-standard ERC20 tokens. This way, the report is presented within a consistent theme.

This perspective is resolutely maintained during the entire project audit, if we were to evaluate it in this way, issue number 545 would need to be divided into 33 separate parts.

novaman33

I do see your point. However, by placing all these issue under one dupe is unfair to watsons and also could deceive the developers. For example one of the dupes is that fee-on-transfer tokens will cause insolvency in the protocol. I saw some functions that track the balance delta between transfers so the developer gave it very high priority for them to be supported. And I see now they have placed a won't fix tag. I also do not agree that this exact issue about those tokens can be medium. Furthermore, I believe that such grouping creates a bad example for future contest. All those issues also have different fixes, which will make it more difficult for the developer to mitigate those issues as they would have to go through all the dupes. Once again I see your point and I do appreciate the good job you did in judging this contest.

0xSmartContract

These issues, grouped under the heading of “Weird ERC20 Tokens,” arise from different token behaviors that arise due to the flexibility of the ERC20 standard.

Addressing such issues separately may require a specific review of each function and contract. However, grouping under specific headings allows us to better understand the root of these issues and develop a general solution strategy. For example, the “fee-on-transfer” and “rebasing” issues can be resolved by dynamically monitoring and appropriately handling transfers and balance changes. This approach can be applied across the protocol and provides a more consistent solution.

The severity of these issues may vary depending on which functions of the protocol they affect and the potential impact of these functions on users. However, making a general grouping allows us to get to the root of these problems and

develop a consistent solution strategy. This helps developers to handle such problems more easily and effectively.

If there were no weird token problems in the entire project (all contracts and all functions), it would be necessary to separate them, but here we are talking about all weird token problems in the entire project

I don't see any effective argument to change the duplicate in 545, my opinion is very clear

jsmi0703

Root cause of #545: non-supporting of fee-on-transfer or rebasing tokens. Root cause of #5, #112, #188, #721: the mistake on supporting fee-on-transfer tokens at `addToPosition()`. If those(#5, #112, #188, #721) not exist, the protocol team couldn't fix the error at all, I think. It is because the protocol team believes that `addToPosition()` already supports fee-on-transfer tokens. Therefore if those not exist, the protocol team could fix fee-on-transfer token problems only in other functions not in `addToPosition()` function. So, they should be grouped as another issue.

0xSmartContract

https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545#issuecomment-2251178050

https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/545#issuecomment-2266711126

In the two detailed opinions above, I have technically detailed why it should be grouped this way

Your list is very weak, incomplete and based on few arguments. I expect you to analyze the entire list and present a detailed argument

Also, if we look at this issue from different effects and angles, we need to make 33 differens issues;

novaman33

Hello, I will not go through all the 80 different issues. I have stated my opinion that these duplications are too general and I still stand by it. I reviewed a few of the issue and it seems like this general approach has caused some of the duplicates to be overlooked. For example I was surprised to see issues like this one - https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/475 marked as valid dupes. I think that many issues here lack sufficient proof of concept and do not show a clear attack path. I will leave it up to the sherlock judge to decide.

0xSmartContract

https://gist.github.com/0xSmartContract/4e7e9fc500f7d6e6061d94ffde6c2206

General approaches for issue #545 and its duplicates do not lead us to a conclusion, if you share your analysis with a list like in the link above, it can be evaluated accordingly

There is a very detailed analysis above, this analysis - despite the listing and detail; "This grouping is too general" does not go beyond a small suggestion

novaman33

Hello, I will not review 80 issues to point out the obvious validations of insufficient reports and duplications with different root causes.If @WangSecurity thinks that my escalation is wrong or does not give enough information,then it can be rejected.

jsmi0703

@0xSmartContract Please read and answer the above comment.

WangSecurity

@jsmi0703 I believe this comment from @0xSmartContract is the answer to your question.

About the issue and the escalation, I believe arguments from both side are completely valid, but this time I'll agree with the Lead Judge and will keep all the issues inside one family based on the following duplication rule:

> If the following issues appear in multiple places, even in different contracts. In that case, they may be considered to have the same root cause: Issues with the same conceptual mistake.
>
> > Example: different untrusted external admins can steal funds

Here, the conceptual mistake is Weird tokens. Hence, planning to reject the escalation and leave the issue as it is.

FYI, I'm asking for this decision to never be quoted or used as reference anywhere as it's an exceptional case and I believe the Lead Judge made a fair decision.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- novaman33: rejected

WangSecurity

Based on https://github.com/sherlock-audit/2024-06-magicsea-judging/issues/650#issuecomment-2294043026 and https://github.com/sherlock-audit/2024-06-m

agicsea-judging/issues/650#issuecomment-2290595068 #545 will be duplicated with #237 with high severity.

WangSecurity

UPD: ignore the above comment, it's irrelevant. For details, take a look at the discussion under #237

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.