SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Convergence
Prepared by:	Sherlock
Lead Security Expert:	0x52
Dates Audited:	November 15 - November 29, 2023
Prepared on:	January 21, 2024

Introduction

Next generation Governance Black Hole. Boosting yields across DeFi, starting with the CurveFinance ecosystem.

Scope

Repository: Cvg-Finance/sherlock-cvg

Branch: main

Commit: d0b36ce5ebb141895e4bf23b241a184fa0606b1b

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
6	3

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

cergyk	lemonmon	GimelSec
hash	0x52	bitsurfer
bughuntoor	detectiveking	0xDetermination

ZanyBonzy
r0ck3tz
cawfree
lil.eth
pontifex
ydlee
vvv
rvierdiiev
jah
0xAlix2
0xkaden
caventa
FarmerRick
Bauer
cducrest-brainbot
CL001

Issue H-1: Lowering the gauge weight can disrupt accounting, potentially leading to both excessive fund distribution and a loss of funds.

Source:
https://github.com/sherlock-audit/2023-11-convergence-judging/issues/94

Found by

0xDetermination, ZanyBonzy, bitsurfer, bughuntoor, hash

Summary

Similar issues were found by users 0xDetermination and bart1e in the Canto veRWA audit, which uses a similar gauge controller type.

Vulnerability Detail

- When the _change_gauge_weight function is called, the `points_weight[addr][next_time].bias` and`time_weight[addr]` are updated - the slope is not.
- The equation ***f(t) = c - mx*** represents the gauge's decay equation before the weight is reduced. In this equation, `m` is the slope. After the weight is reduced by an amount `k` using the `change_gauge_weight` function, the equation becomes ***f(t) = c - k - mx*** The slope m remains unchanged, but the t-axis intercept changes from ***t1 = c/m*** to ***t2 = (c-k)/m***.
- Slope adjustments that should be applied to the global slope when decay reaches 0 are stored in the `changes_sum` hashmap. And is not affected by changes in gauge weight. Consequently, there's a time window ***t1 - t2*** during which the earlier slope changes applied to the global state when user called `vote_for_gauge_weights` function remains applied even though they should have been subtracted. This in turn creates a situation in which the global weightis less than the sum of the individual gauge weights, resulting in an accounting error.
- So, in the `CvgRewards` contract when the `writeStakingRewards` function invokes the `_checkpoint`, which subsequently triggers the `gauge_relative_weight_writes` function for the relevant time period, the calculated relative weight becomes inflated, leading to an increase in the distributed rewards. If all available rewards are distributed before the entire array is processed, the remaining users will receive no rewards."

- The issue mainly arises when a gauge's weight has completely diminished to zero. This is certain to happen if a gauge with a non-zero bias, non-zero slope, and a t-intercept exceeding the current time is killed using `kill_gauge` function.
- Additionally, decreasing a gauge's weight introduces inaccuracies in its decay equation, as is evident in the t-intercept.

Impact

The way rewards are calculated is broken, leading to an uneven distribution of rewards, with some users receiving too much and others receiving nothing.

Code Snippet

https://github.com/sherlock-audit/2023-11-convergence/blob/e894be3e36614a385cf409dc7e278d5b8f16d6f2/sherlock-cvg/contracts/Locking/GaugeController.vy#L568C1-L590C1

https://github.com/sherlock-audit/2023-11-convergence/blob/e894be3e36614a385cf409dc7e278d5b8f16d6f2/sherlock-cvg/contracts/Rewards/CvgRewards.sol#L189

https://github.com/sherlock-audit/2023-11-convergence/blob/e894be3e36614a385cf409dc7e278d5b8f16d6f2/sherlock-cvg/contracts/Rewards/CvgRewards.sol#L235C1-L235C91

https://github.com/sherlock-audit/2023-11-convergence/blob/e894be3e36614a385cf409dc7e278d5b8f16d6f2/sherlock-cvg/contracts/Locking/GaugeController.vy#L493

https://github.com/sherlock-audit/2023-11-convergence/blob/e894be3e36614a385cf409dc7e278d5b8f16d6f2/sherlock-cvg/contracts/Locking/GaugeController.vy#L456C1-L475C17

https://github.com/sherlock-audit/2023-11-convergence/blob/e894be3e36614a385cf409dc7e278d5b8f16d6f2/sherlock-cvg/contracts/Locking/GaugeController.vy#L603C1-L611C54

Tool used

Manual Review

Recommendation

Disable weight reduction, or only allow reset to 0.

Discussion

0xR3vert

Hello,

Thanks a lot for your attention.

Thank you for your insightful observation. Upon thorough examination, we acknowledge that such an occurrence could indeed jeopardize the protocol. We are currently exploring multiple solutions to address this issue. We are considering removing the function change_gauge_weight entirely and not distributing CVG inflation on killed gauges, similar to how Curve Protocol handles their gauges.

Therefore, in conclusion, we must consider your issue as valid.

Regards, Convergence Team

CergyK

Escalate

The severity of this issue is low for two reasons:

- Admin endpoint, the admin can be trusted to not use this feature lightly, and take preventative measures to ensure that accounting is not disrupted, such as ensuring that there is no current votes for a gauge and locking voting to set a weight.
- _change_gauge_weight has this exact implementation in the battle-tested curve dao contract (in usage for more than 3 years now without notable issue): https://github.com/curvefi/curve-dao-contracts/blob/master/contracts/GaugeController.vy

sherlock-admin2

> Escalate
>
> The severity of this issue is low for two reasons:
>
> - Admin endpoint, the admin can be trusted to not use this feature lightly, and take preventative measures to ensure that accounting is not disrupted, such as ensuring that there is no current votes for a gauge and locking voting to set a weight.
> - _change_gauge_weight has this exact implementation in the battle-tested curve dao contract (in usage for more than 3 years now without notable issue): https://github.com/curvefi/curve-dao-contracts/blob/master/contracts/GaugeController.vy

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.

0xDetermination

Addressing the escalation points:

1. If there are no current votes for a gauge, the weight can't be lowered. Also, locking voting is not really relevant for this issue. I don't think there are any preventative measures that can be taken other than never lowering the gauge weight.
2. This function is in the live Curve contract, but it has never been called (see

   nevillehuang

   I think all issues regarding killing and changing weight for gauges (#18, #94, #122,#192), all the arguments are assuming the following:

   (a) The admin would perform appropriate measures before executing admin gated functions - To me, this is not clear cut like the way admin input would be. From sponsor confirmation, you would understand that they did not understand the severity of performing such issues so it wouldn't be unreasonable to say they are not aware enough to perform such preventive measures before this functions are called. If not, I think this should have been explicitly mentioned in known design considerations in the contest details and/or the contract details here, where the purpose of locking and pausing votes are stated.

   (b) Afaik, when @CergyK mentions the admin can take preventive measures such as ensuring no current vote and locking votes, then what would happen during a scenario when the current gauge that an admin wants to change weight or kill gauge (e.g. malicious token) has votes assigned. Wouldn't that essentially mean admin can never do so, and therefore breaks core functionality?

   (c) The admin would never call `change_gauge_weight` because it has never been called in a live curve contract - This is pure speculation, just because curve doesn't call it, it does not mean convergence would never call it.

   Czar102

   See my comment here.

   Planning to reject the escalation.

   nevillehuang

   @0xDetermination @deadrosesxyz @10xhash

Do you think this is a duplicate of #192 because they seem similar. I am unsure if fixing one issue will fix another, given at the point of contest, it is intended to invoke both functions.

0xDetermination

@nevillehuang I think any issue with a root cause of 'lowering gauge weight' should be considered the same if I understand the duplication rules correctly. So it seems like these are all dupes.

Czar102

As long as an issue is valid and the root cause is the same, they are currently considered duplicates. Both #192 and #94 have a root cause of not handling the slope in `_change_gauge_weight`, despite having different impacts.

Czar102

Result: High Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- CergyK: rejected

walk-on-me

Hello, we fixed this issue on this PR.

You can see on these comments, description of the fix :

- https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457713428
- https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457714042

IAm0x52

Fix looks good. _change_gauge_weight has been removed completely

Issue H-2: LockingPositionDelegate::manageOwnedAndDeleg unchecked duplicate tokenId allow metaGovernance manipulation

Source:
https://github.com/sherlock-audit/2023-11-convergence-judging/issues/126

Found by

bughuntoor, cawfree, cergyk, hash, lemonmon, r0ck3tz

Summary

A malicious user can multiply his share of meta governance delegation for a tokenId by adding that token multiple times when calling `manageOwnedAndDelegated`

Vulnerability Detail

Without checks to prevent the addition of duplicate token IDs, a user can artificially inflate their voting power and their metaGovernance delegations.

A malicious user can add the same tokenId multiple times, and thus multiply his own share of meta governance delegation with regards to that tokenId.

Scenario:

(a) Bob delegates a part of metaGovernance to Mallory - he allocates 10% to her and 90% to Alice.

(b) Mallory calls `manageOwnedAndDelegated` and adds the same `tokenId` 10 times, each time allocating 10% of the voting power to herself.

(c) Mallory now has 100% of the voting power for the `tokenId`, fetched by calling `mgCvgVotingPowerPerAddress`, harming Bob and Alice metaGovernance voting power.

Impact

The lack of duplicate checks can be exploited by a malicious user to manipulate the metaGovernance system, allowing her to gain illegitimate voting power (up to 100%) on a delegated tokenId, harming the delegator and the other delegations of the same `tokenId`.

Code Snippet

```
    function manageOwnedAndDelegated(OwnedAndDelegated calldata
↪   _ownedAndDelegatedTokens) external {
        ...

      for (uint256 i; i < _ownedAndDelegatedTokens.owneds.length;) {
↪   //@audit no duplicate check
        ...
        }

      for (uint256 i; i < _ownedAndDelegatedTokens.mgDelegateds.length;) {
↪   //@audit no duplicate check
        ...
        }

      for (uint256 i; i < _ownedAndDelegatedTokens.veDelegateds.length;) {
↪   //@audit no duplicate check
        ...
        }
    }
```

```
function mgCvgVotingPowerPerAddress(address _user) public view returns
↪   (uint256) {
        uint256 _totalMetaGovernance;
        ...
        /** @dev Sum voting power from delegated (allowed) tokenIds to
↪   _user. */
        for (uint256 i; i < tokenIdsDelegateds.length; ) {
            uint256 _tokenId = tokenIdsDelegateds[i];
            (uint256 _toPercentage, , uint256 _toIndex) =
↪   _lockingPositionDelegate.getMgDelegateeInfoPerTokenAndAddress(
                _tokenId,
                _user
            );
            /** @dev Check if is really delegated, if not mg voting power
↪   for this tokenId is 0. */
            if (_toIndex < 999) {
                uint256 _tokenBalance = balanceOfMgCvg(_tokenId);
                _totalMetaGovernance += (_tokenBalance * _toPercentage) /
↪   MAX_PERCENTAGE;
            }

            unchecked {
                ++i;
            }
        }
        ...
    return _totalMetaGovernance; //@audit total voting power for the
↪   tokenID which will be inflated by adding the same tokenID multiple times
    }
```

https://github.com/sherlock-audit/2023-11-convergence/blob/main/sherlock-cvg/contracts/Locking/LockingPositionDelegate.sol#L330

PoC

Add in balance-delegation.spec.ts:

```
it("Manage tokenIds for user10 with dupes", async () => {
    let tokenIds = {owneds: [], mgDelegateds: [1, 1], veDelegateds: []};
    await lockingPositionDelegate.connect(user10).manageOwnedAndDelegated(t⌋
↪   okenIds);
});

it("Checks mgCVG balances of user10 (delegatee)", async () => {
    const tokenBalance = await lockingPositionService.balanceOfMgCvg(1);

    // USER 10
    const delegatedPercentage = 70n;

    //@audit: The voting power is multiplied by 2 due to the duplicate
    const exploit_multiplier = 2n;
    const expectedVotingPower = (exploit_multiplier * tokenBalance *
↪   delegatedPercentage) / 100n;
    const votingPower = await
↪   lockingPositionService.mgCvgVotingPowerPerAddress(user10);

    // take solidity rounding down into account
    expect(votingPower).to.be.approximately(expectedVotingPower, 1);
});
```

Tool used

Recommendation

Ensuring the array of token IDs is sorted and contains no duplicates. This can be achieved by verifying that each tokenId in the array is strictly greater than the previous one, it ensures uniqueness without additional data structures.

Discussion

shalbe-cvg

Hello,

Thanks a lot for your attention.

After an in-depth review, we have to consider your issue as Confirmed. We will add a check on the values contained in the 3 arrays to ensure duplicates are taken away before starting the process.

Regards, Convergence Team

walk-on-me

Hello dear auditor,

We performed the correction of this issue.

We used the trick you give us to check the duplicates in the arrays of token ID.

You can find the correction here :

https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457545377 https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457546051 https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457546527

IAm0x52

Fix looks good. Arrays that have duplicates or that aren't ordered will cause the function to revert

Issue H-3: Tokens that are both bribes and StakeDao gauge rewards will cause loss of funds

Source:
https://github.com/sherlock-audit/2023-11-convergence-judging/issues/182

Found by

0x52, GimelSec, detectiveking, lemonmon

Summary

When SdtStakingPositionService is pulling rewards and bribes from buffer, the buffer will return a list of tokens and amounts owed. This list is used to set the rewards eligible for distribution. Since this list is never check for duplicate tokens, a shared bribe and reward token would cause the token to show up twice in the list. The issue it that _sdtRewardsByCycle is set and not incremented which will cause the second occurrence of the token to overwrite the first and break accounting. The amount of token received from the gauge reward that is overwritten will be lost forever.

Vulnerability Detail

In L559 of SdtStakingPositionService it receives a list of tokens and amount from the buffer.

SdtBuffer.sol#L90-L168

```
ICommonStruct.TokenAmount[] memory bribeTokens =
↪   _sdtBlackHole.pullSdStakingBribes(
    processor,
    _processorRewardsPercentage
);

uint256 rewardAmount = _gaugeAsset.reward_count();

ICommonStruct.TokenAmount[] memory tokenAmounts = new
↪   ICommonStruct.TokenAmount[](
    rewardAmount + bribeTokens.length
);

uint256 counter;
address _processor = processor;
for (uint256 j; j < rewardAmount; ) {
    IERC20 token = _gaugeAsset.reward_tokens(j);
    uint256 balance = token.balanceOf(address(this));
    if (balance != 0) {
        uint256 fullBalance = balance;

        ...

        token.transfer(sdtRewardsReceiver, balance);

      **@audit token and amount added from reward_tokens pulled directly
       ↪  from gauge**

        tokenAmounts[counter++] = ICommonStruct.TokenAmount({token: token,
         ↪  amount: balance});
    }

    ...

}

for (uint256 j; j < bribeTokens.length; ) {
    IERC20 token = bribeTokens[j].token;
    uint256 amount = bribeTokens[j].amount;

  **@audit token and amount added directly with no check for duplicate
   ↪  token**

    if (amount != 0) {
        tokenAmounts[counter++] = ICommonStruct.TokenAmount({token: token,
         ↪  amount: amount});

    ...

}
```

SdtBuffer#pullRewards returns a list of tokens that is a concatenated array of all bribe and reward tokens. There is not controls in place to remove duplicates from this list of tokens. This means that tokens that are both bribes and rewards will be duplicated in the list.

SdtStakingPositionService.sol#L561-L577

```
for (uint256 i; i < _rewardAssets.length; ) {
    IERC20 _token = _rewardAssets[i].token;
    uint256 erc20Id = _tokenToId[_token];
    if (erc20Id == 0) {
        uint256 _numberOfSdtRewards = ++numberOfSdtRewards;
        _tokenToId[_token] = _numberOfSdtRewards;
        erc20Id = _numberOfSdtRewards;
    }

  **@audit overwrites and doesn't increment causing duplicates to be lost**

    _sdtRewardsByCycle[_cvgStakingCycle][erc20Id] =
     ↪  ICommonStruct.TokenAmount({
        token: _token,
        amount: _rewardAssets[i].amount
    });
    unchecked {
        ++i;
    }
}
```

When storing this list of rewards, it overwrites _sdtRewardsByCycle with the values from the returned array. This is where the problem arises because duplicates will cause the second entry to overwrite the first entry. Since the first instance is overwritten, all funds in the first occurrence will be lost permanently.

Impact

Tokens that are both bribes and rewards will be cause tokens to be lost forever

Code Snippet

SdtStakingPositionService.sol#L550-L582

Tool used

Manual Review

Recommendation

Either sdtBuffer or SdtStakingPositionService should be updated to combine duplicate token entries and prevent overwriting.

Discussion

0xR3vert

Hello,

Thanks a lot for your attention.

Absolutely, if we kill a gauge or change a type weight during the distribution, it would distribute wrong amounts, even though we're not planning to do that. We can make sure it doesn't happen by doing what you said: locking those functions to avoid any problems.

Therefore, in conclusion, we must consider your issue as a valid.

Regards, Convergence Team

walk-on-me

This issue has been solved here :

https://github.com/Cvg-Finance/sherlock-cvg/pull/4

Follow the comment : https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457470558

IAm0x52

Fix looks good. Now uses a sum instead of a set

Issue M-1: Division by Zero in CvgRewards::_distributeCvgRewards leads to locked funds

Source:
https://github.com/sherlock-audit/2023-11-convergence-judging/issues/131

Found by

cergyk

Summary

The bug occurs when `CvgRewards::_setTotalWeight` sets `totalWeightLocked` to zero, leading to a division by zero error in `CvgRewards::_distributeCvgRewards`, and blocking cycle increments. The blocking results in all Cvg locked to be unlockable permanently.

Vulnerability Detail

The function `_distributeCvgRewards` of `CvgRewards.sol` is designed to calculate and distribute CVG rewards among staking contracts. It calculates the `cvgDistributed` for each gauge based on its weight and the total staking inflation. However, if the `totalWeightLocked` remains at zero (due to some gauges that are available but no user has voted for any gauge), the code attempts to divide by zero.

The DoS of `_distributeCvgRewards` will prevent cycle from advancing to the next state `State.CONTROL_TOWER_SYNC`, thus forever locking the users' locked CVG tokens.

Impact

Loss of users' CVG tokens due to DoS of `_distributeCvgRewards` blocking the state.

Code Snippet

```
    function _setTotalWeight() internal {
        ...
      totalWeightLocked +=
↪   _gaugeController.get_gauge_weight_sum(_getGaugeChunk(_cursor,
↪   _endChunk)); //@audit `totalWeightLocked` can be set to 0 if no gauge
↪   has received any vote
        ...
    }

```

```
    function _distributeCvgRewards() internal {
        ...
        uint256 _totalWeight = totalWeightLocked;
        ...
        for (uint256 i; i < gaugeWeights.length; ) {
            /// @dev compute the amount of CVG to distribute in the gauge
          cvgDistributed = (stakingInflation * gaugeWeights[i]) /
 ↪  _totalWeight; //@audit will revert if `_totalWeight` is zero
        ...

```

```
/**
* @notice Unlock CVG tokens under the NFT Locking Position : Burn the NFT,
 ↪  Transfer back the CVG to the user.  Rewards from YsDistributor must be
 ↪  claimed before or they will be lost.    * @dev The locking time must be
 ↪  over
* @param tokenId to burn
*/
function burnPosition(uint256 tokenId) external {
...
      require(_cvgControlTower.cvgCycle() > lastEndCycle, "LOCKED");
 ↪  //@audit funds are locked if current `cycle <= lastEndCycle`
...
    }
```

https://github.com/sherlock-audit/2023-11-convergence/blob/main/sherlock-cvg/contracts/Rewards/CvgRewards.sol#L321

Tool used

Recommendation

If the _totalWeight is zero, just consider the cvg rewards to be zero for that cycle, and continue with other logic:

```
-cvgDistributed = (stakingInflation * gaugeWeights[i]) / _totalWeight;
+cvgDistributed = _totalWeight == 0 ? 0 : (stakingInflation *
 ↪  gaugeWeights[i]) / _totalWeight;
```

Discussion

0xR3vert

Hello,

Thanks a lot for your attention.

We are aware of the potential for a division by zero if there are no votes at all in one of our gauges. However, this scenario is unlikely to occur in reality because there will always be votes deployed (by us and/or others) in the gauges. Nevertheless, your point is valid, and we will address it to be prepared for this case.

Therefore, in conclusion, we must acknowledge your issue as correct, even though we are already aware of it.

Regards, Convergence Team

nevillehuang

Since DoS is not permanent where in as long as protocol/users themselves vote for the gauge, I think this is a low severity issue.

CergyK

Escalate

Escalating based on latest comment:

> Since DoS is not permanent where in as long as protocol/users themselves vote for the gauge, I think this is a low severity issue.

If we reach this case (`totalWeights == 0`), the DoS is permanent. There would be no other way to reset this variable, and all user funds would be locked permanently.

It is acknowledged that there is a low chance of this happening, but due to the severe impact and acknowledged validity this should be a medium

sherlock-admin2

> Escalate
>
> Escalating based on latest comment:
>
>> Since DoS is not permanent where in as long as protocol/users themselves vote for the gauge, I think this is a low severity issue.
>
> If we reach this case (`totalWeights == 0`), the DoS is permanent. There would be no other way to reset this variable, and all user funds would be locked permanently.

> It is acknowledged that there is a low chance of this happening, but due to the severe impact and acknowledged validity this should be a medium

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.

nevillehuang

@CergyK As mentioned by the sponsor, they will always ensure there is votes present to prevent this scenario, so I can see this as an "admin error" if the scenario is allowed to happen, but I also see your point given this was not made known to watsons. If totalWeight goes to zero, it will indeed be irrecoverable.

Unlikely but possible, so can be valid based on this sherlock rule

> Causes a loss of funds but requires certain external conditions or specific states.

Czar102

I believe this impact warrants medium severity. Planning to accept the escalation.

Czar102

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- CergyK: accepted

walk-on-me

Hello dear auditor,

we fixed this issue.

You can find how on the following link :

https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457528119

IAm0x52

Fix looks good. Now utilizes a ternary operator to prevent division by zero

Issue M-2: LockPositionService::increaseLockTime Incorrect Calculation Extends Lock Duration Beyond Intended Period

Source:
https://github.com/sherlock-audit/2023-11-convergence-judging/issues/136

Found by

bughuntoor, cergyk, jah, rvierdiiev

Summary

`LockPositionService::increaseLockTime` uses `block.timestamp` for locking tokens, resulting in potential over-extension of the lock period. Specifically, if a user locks tokens near the end of a cycle, the lock duration might extend an additional week more than intended. For instance, locking for one cycle at the end of cycle N could result in an unlock time at the end of cycle N+2, instead of at the start of cycle N+2.

This means that all the while specifying that their $CVG should be locked for the next cycle, the $CVG stays locked for two cycles.

Vulnerability Detail

The function `increaseLockTime` inaccurately calculates the lock duration by using `block.timestamp`, thus not aligned to the starts of cycles. This discrepancy leads to a longer-than-expected lock period, especially when a lock is initiated near the end of a cycle. This misalignment means that users are unintentionally extending their lock period and affecting their asset management strategies.

Scenario:

- Alice decides to lock her tokens for one cycle near the end of cycle N.
- The lock duration calculation extends the lock to the end of cycle N+2, rather than starting the unlock process at the start of cycle N+2.
- Alice's tokens are locked for an additional week beyond her expectation.

Impact

Code Snippet

https://github.com/sherlock-audit/2023-11-convergence/blob/main/sherlock-cvg/contracts/Locking/LockingPositionService.sol#L421

Tool used

Users may have their $CVG locked for a week more than expected

Recommendation

Align the locking mechanism to multiples of a week and use `(block.timestamp % WEEK) + lockDuration` for the lock time calculation. This adjustment ensures that the lock duration is consistent with user expectations and cycle durations.

Discussion

walk-on-me

Hello dear judge,

this issue has been solved.

In order to solve it, we decided that when a user comes for locking or increase it's lock, we'll not take anymore the one in the CvgControlTower. If an user decide to performs a lock action between the timestamp where the cycle is updated and the CVG distribution (by the CvgRewards). He'll just lock for the next week, having no incidence on protocol.

Remarks : We are keeping the cvgCycle of the CvgControlTower for :

Burning the NFT Claim the rewards of the Ys You can check how by following the comments done here :

https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457499501 https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457504596 https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457504988 https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457505267 https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457508720
https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457510511

IAm0x52

Fix looks good. Cycles are now aligned based on computed week rather than cvgControlTower

Issue M-3: Delegation Limitation in Voting Power Management

Source:
https://github.com/sherlock-audit/2023-11-convergence-judging/issues/142

Found by

lil.eth, pontifex, ydlee

Summary

MgCVG Voting power delegation system is constrained by 2 hard limits, first on the number of tokens delegated to one user (`maxTokenIdsDelegated = 25`) and second on the number of delegatees for one token (`maxMgDelegatees = 5`). Once this limit is reached for a token, the token owner cannot modify the delegation percentage to an existing delegated user. This inflexibility can prevent efficient and dynamic management of delegated voting power.

Vulnerability Detail

Observe these lines :

```
function delegateMgCvg(uint256 _tokenId, address _to, uint96 _percentage)
↪   external onlyTokenOwner(_tokenId) {
    require(_percentage <= 100, "INVALID_PERCENTAGE");

    uint256 _delegateesLength = delegatedMgCvg[_tokenId].length;
    require(_delegateesLength < maxMgDelegatees, "TOO_MUCH_DELEGATEES");

    uint256 tokenIdsDelegated = mgCvgDelegatees[_to].length;
    require(tokenIdsDelegated < maxTokenIdsDelegated,
↪   "TOO_MUCH_MG_TOKEN_ID_DELEGATED");
```

if either `maxMgDelegatees` or `maxTokenIdsDelegated` are reached, delegation is no longer possible. The problem is the fact that this function can be either used to delegate or to update percentage of delegation or also to remove a delegation but in cases where we already delegated to a maximum of users (`maxMgDelegatees`) OR the user to who we delegated has reached the maximum number of tokens that can be delegated to him/her (`maxTokenIdsDelegated`), an update or a removal of delegation is no longer possible.

6 scenarios are possible :

(a) `maxTokenIdsDelegated` is set to 5, Alice is the third to delegate her voting power to Bob and choose to delegate 10% to him. Bob gets 2 other people delegating their tokens to him, Alice wants to increase the power delegated to Bob to 50% but she cannot due to Bob reaching `maxTokenIdsDelegated`

(b) `maxTokenIdsDelegated` is set to 25, Alice is the 10th to delegate her voting power to Bob and choose to delegate 10%, DAO decrease `maxTokenIdsDelegated` to 3, Alice wants to increase the power delegated to Bob to 50%, but she cannot due to this

(c) `maxTokenIdsDelegated` is set to 5, Alice is the third to delegate her voting power to Bob and choose to delegate 90%. Bob gets 2 other people delegating their tokens to him, Alice wants to only remove the power delegated to Bob using this function, but she cannot due to this

(d) `maxMgDelegatees` is set to 3, Alice delegates her voting power to Bob,Charly and Donald by 20% each, Alice reaches `maxMgDelegatees` and she cannot update her voting power for any of Bob,Charly or Donald

(e) `maxMgDelegatees` is set to 5, Alice delegates her voting power to Bob,Charly and Donald by 20% each,DAO decreases`maxMgDelegatees` to 3. Alice cannot update or remove her voting power delegated to any of Bob,Charly and Donald

(f) `maxMgDelegatees` is set to 3, Alice delegates her voting power to Bob,Charly and Donald by 20% each, Alice wants to only remove her delegation to Bob but she reached `maxMgDelegatees` so she cannot only remove her delegation to Bob

A function is provided to remove all user to who we delegated but this function cannot be used as a solution to this problem due to 2 things :

- It's clearly not intended to do an update of voting power percentage by first removing all delegation we did because `delegateMgCvg()` is clearly defined to allow to delegate OR to remove one delegation OR to update percentage of delegation but in some cases it's impossible which is not acceptable
- if Alice wants to update it's percentage delegated to Bob , she would have to remove all her delegatees and would take the risk that someone is faster than her and delegate to Bob before her, making Bob reaches `maxTokenIdsDelegated` and would render impossible for Alice to re-delegate to Bob

POC

You can add it to test/ut/delegation/balance-delegation.spec.ts :

```
it("maxTokenIdsDelegated is reached => Cannot update percentage of
↪  delegate", async function () {
        (await
↪  lockingPositionDelegate.maxTokenIdsDelegated()).should.be.equal(25);
        await
↪  lockingPositionDelegate.connect(treasuryDao).setMaxTokenIdsDelegated(3);
        (await
↪  lockingPositionDelegate.maxTokenIdsDelegated()).should.be.equal(3);

        await lockingPositionDelegate.connect(user1).delegateMgCvg(1,
↪  user10, 20);
        await lockingPositionDelegate.connect(user2).delegateMgCvg(2,
↪  user10, 30);
        await lockingPositionDelegate.connect(user3).delegateMgCvg(3,
↪  user10, 30);

        const txFail =
↪  lockingPositionDelegate.connect(user1).delegateMgCvg(1, user10, 40);
        await
↪  expect(txFail).to.be.revertedWith("TOO_MUCH_MG_TOKEN_ID_DELEGATED");
    });
    it("maxTokenIdsDelegated IS DECREASED => PERCENTAGE UPDATE IS NO LONGER
↪  POSSIBLE", async function () {
        await lockingPositionDelegate.connect(treasuryDao).setMaxTokenIdsDe⌋
↪  legated(25);
        (await
↪  lockingPositionDelegate.maxTokenIdsDelegated()).should.be.equal(25);

        await lockingPositionDelegate.connect(user1).delegateMgCvg(1,
↪  user10, 20);
        await lockingPositionDelegate.connect(user2).delegateMgCvg(2,
↪  user10, 30);
        await lockingPositionDelegate.connect(user3).delegateMgCvg(3,
↪  user10, 30);

        await
↪  lockingPositionDelegate.connect(treasuryDao).setMaxTokenIdsDelegated(3);
        (await
↪  lockingPositionDelegate.maxTokenIdsDelegated()).should.be.equal(3);

        const txFail =
↪  lockingPositionDelegate.connect(user1).delegateMgCvg(1, user10, 40);
        await
↪  expect(txFail).to.be.revertedWith("TOO_MUCH_MG_TOKEN_ID_DELEGATED");
        await lockingPositionDelegate.connect(treasuryDao).setMaxTokenIdsDe⌋
↪  legated(25);
```

```
        (await
↪   lockingPositionDelegate.maxTokenIdsDelegated()).should.be.equal(25);
    });
    it("maxMgDelegatees : TRY TO UPDATE PERCENTAGE DELEGATED TO A USER IF
↪   WE ALREADY REACH maxMgDelegatees", async function () {
        await
↪   lockingPositionDelegate.connect(treasuryDao).setMaxMgDelegatees(3);
        (await
↪   lockingPositionDelegate.maxMgDelegatees()).should.be.equal(3);

        await lockingPositionDelegate.connect(user1).delegateMgCvg(1,
↪   user10, 20);
        await lockingPositionDelegate.connect(user1).delegateMgCvg(1,
↪   user2, 30);
        await lockingPositionDelegate.connect(user1).delegateMgCvg(1,
↪   user3, 30);

        const txFail =
↪   lockingPositionDelegate.connect(user1).delegateMgCvg(1, user10, 40);
        await expect(txFail).to.be.revertedWith("TOO_MUCH_DELEGATEES");
    });
    it("maxMgDelegatees : maxMgDelegatees IS DECREASED => PERCENTAGE UPDATE
↪   IS NO LONGER POSSIBLE", async function () {
        await
↪   lockingPositionDelegate.connect(treasuryDao).setMaxMgDelegatees(5);
        (await
↪   lockingPositionDelegate.maxMgDelegatees()).should.be.equal(5);

        await lockingPositionDelegate.connect(user1).delegateMgCvg(1,
↪   user10, 20);
        await lockingPositionDelegate.connect(user1).delegateMgCvg(1,
↪   user2, 30);
        await lockingPositionDelegate.connect(user1).delegateMgCvg(1,
↪   user3, 10);

        await
↪   lockingPositionDelegate.connect(treasuryDao).setMaxMgDelegatees(2);
        (await
↪   lockingPositionDelegate.maxMgDelegatees()).should.be.equal(2);

        const txFail2 =
↪   lockingPositionDelegate.connect(user1).delegateMgCvg(1, user2, 50);
        await expect(txFail2).to.be.revertedWith("TOO_MUCH_DELEGATEES");
    });
```

Impact

In some cases it is impossible to update percentage delegated or to remove only one delegated percentage then forcing users to remove all their voting power delegatations, taking the risk that someone is faster then them to delegate to their old delegated users and reach threshold for delegation, making impossible for them to re-delegate

Code Snippet

https://github.com/sherlock-audit/2023-11-convergence/blob/main/sherlock-cvg/contracts/Locking/LockingPositionDelegate.sol#L278

Tool used

Manual Review

Recommendation

Separate functions for new delegations and updates : Implement logic that differentiates between adding a new delegatee and updating an existing delegation to allow updates to existing delegations even if the maximum number of delegatees is reached

Discussion

shalbe-cvg

Hello,

Thanks a lot for your attention.

After an in-depth review, we have to consider your issue as Invalid. We have developed a function allowing users to clean their `mgDelegatees` and `veDelegatees`. Therefore there is no need to divide this delegation function into two different ones (add / update).

Regards, Convergence Team

nevillehuang

Hi @0xR3vert @shalbe-cvg @walk-on-me,

Could point me to the existing function you are pointing to that is present during the time of the audit?

shalbe-cvg

> Hi @0xR3vert @shalbe-cvg @walk-on-me,
>
> Could point me to the existing function you are pointing to that is present during the time of the audit?

Hello, this is the function `cleanDelegatees` inside LockingPositionDelegate contract

nevillehuang

Agree with sponsor, since `cleanDelegatees()` and `removeTokenIdDelegated()` here allow removal of delegatees one-by-one, this seems to be a non-issue.

ChechetkinVV

Escalate

> Agree with sponsor, since `cleanDelegatees()` and `removeTokenIdDelegated()` here allow removal of delegatees one-by-one, this seems to be a non-issue.

The `cleanDelegatees` function referred to by the sponsor allows the owner of the token to completely remove delegation from ALL mgDelegates, but it will not be possible to remove delegation from ONE delegate using this function. This is obvious from the _cleanMgDelegatees function which is called from the cleanDelegatees function.

The only way for the owner to remove delegation from ONE delegate is using the `delegateMgCvg` function. But this becomes impossible if the delegate from whom the owner is trying to remove delegation has reached the maximum number of delegations.

Perhaps recommendations from https://github.com/sherlock-audit/2023-11-convergence-judging/issues/202 and https://github.com/sherlock-audit/2023-11-convergence-judging/issues/206 reports will help to better understand this problem.

This problem is described in this report and in https://github.com/sherlock-audit/2023-11-convergence-judging/issues/202, https://github.com/sherlock-audit/2023-11-convergence-judging/issues/206 reports. Other reports describe different problems. They are hardly duplicates of this issue.

sherlock-admin2

> Escalate
>
> > Agree with sponsor, since `cleanDelegatees()` and `removeTokenIdDelegated()` here allow removal of delegatees one-by-one, this seems to be a non-issue.

> The `cleanDelegatees` function referred to by the sponsor allows the owner of the token to completely remove delegation from ALL mgDelegates, but it will not be possible to remove delegation from ONE delegate using this function. This is obvious from the _cleanMgDelegatees function which is called from the cleanDelegatees function.
>
> The only way for the owner to remove delegation from ONE delegate is using the `delegateMgCvg` function. But this becomes impossible if the delegate from whom the owner is trying to remove delegation has reached the maximum number of delegations.
>
> Perhaps recommendations from https://github.com/sherlock-audit/2023-11-convergence-judging/issues/202 and https://github.com/sherlock-audit/2023-11-convergence-judging/issues/206 reports will help to better understand this problem.
>
> This problem is described in this report and in https://github.com/sherlock-audit/2023-11-convergence-judging/issues/202, https://github.com/sherlock-audit/2023-11-convergence-judging/issues/206 reports. Other reports describe different problems. They are hardly duplicates of this issue.

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.

nevillehuang

@shalbe-cvg @walk-on-me @0xR3vert @ChechetkinVV

I think I agree with watsons in the sense that it seems not intended to remove all delegations once max delegation number is reached just to adjust voting power or to remove a single delegatee, for both mgCVG and veCVG.

I think what the watsons are mentioning is that this would then open up a scenario of potential front-running for delegatees themselves to permanently always have max delegated mgCVG or veCVG, so it would permanently DoS the original delegator from updating/removing delegatee.

Czar102

From my understanding, this issue presents an issue of breaking core functionality (despite the contracts working according to the design, core intended functionality is impacted).

I believe this is sufficient to warrant medium severity for the issue.

@nevillehuang which issues should and which shouldn't be duplicated with this one? Do you agree with the escalation?

nevillehuang

@Czar102, As I understand, there are two current impacts

(a) Cannot clean delegates one by one, but forced to clean all delegates when `maxMgDelegatees` or `maxTokenIdsDelegated`

(b) Frontrunning to force DOS on delegator performing delegation removal from a delegator to invoke the max delegation check

This is what I think are duplicates:

(a) 142 (this issue mentions both impacts), 202, 206

(b) 40, 51, 142 (again this issue mentions both impacts), 169

Both impact arises from the `maxMgDelegatees/maxTokenIdsDelegated` check thats why I originally grouped them all together. While I initially disagreed, on further review I agree with escalation since this is not the intended contract functionality intended by the protocol. For impact 2, based on your previous comments here, it seems like it is low severity.

Czar102

Thank you for the summary @nevillehuang. I'm planning to consider all other issues (apart from #142, #202, #206) low, hence they will not be considered duplicates anymore (see docs). The three issues describing a Medium severity impact will be considered duplicates and be rewarded.

Czar102

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- ChechetkinVV: accepted

Issue M-4: cvgControlTower and veCVG lock timing will be different and lead to yield loss scenarios

Source:
https://github.com/sherlock-audit/2023-11-convergence-judging/issues/178

Found by

0x52, 0xAlix2, bughuntoor, cergyk, hash

Summary

When creating a locked CVG position, there are two more or less independent locks that are created. The first is in lockingPositionService and the other is in veCVG. LockingPositionService operates on cycles (which are not finite length) while veCVG always rounds down to the absolute nearest week. The disparity between these two accounting mechanism leads to conflicting scenario that the lock on LockingPositionService can be expired while the lock on veCVG isn't (and vice versa). Additionally tokens with expired locks on LockingPositionService cannot be extended. The result is that the token is expired but can't be withdrawn. The result of this is that the expired token must wait to be unstaked and then restaked, cause loss of user yield and voting power while the token is DOS'd.

Vulnerability Detail

Cycles operate using block.timestamp when setting lastUpdateTime on the new cycle in L345. It also requires that at least 7 days has passed since this update to roll the cycle forward in L205. The result is that the cycle can never be exactly 7 days long and the start/end of the cycle will constantly fluctuate.

Meanwhile when veCVG is calculating the unlock time it uses the week rounded down as shown in L328.

We can demonstrate with an example:

Assume the first CVG cycle is started at block.timestamp == 1,000,000. This means our first cycle ends at 1,604,800. A user deposits for a single cycle at 1,400,000. A lock is created for cycle 2 which will unlock at 2,209,600.

The lock on veCVG does not match this though. Instead it's calculation will yield:

```
(1,400,000 + 2 * 604,800) / 604,800 = 4
```

```
4 * 604,800 = 2,419,200
```

As seen these are mismatched and the token won't be withdrawable until much after it should be due to the check in veCVG L404.

This DOS will prevent the expired lock from being unstaked and restaked which causes loss of yield.

The opposite issue can also occur. For each cycle that is slightly longer than expected the veCVG lock will become further and further behind the cycle lock on lockingPositionService. This can also cause a dos and yield loss because it could prevent user from extending valid locks due to the checks in L367 of veCVG.

An example of this:

Assume a user locks for 96 weeks (58,060,800). Over the course of that year, it takes an average of 2 hours between the end of each cycle and when the cycle is rolled over. This effectively extends our cycle time from 604,800 to 612,000 (+7200). Now after 95 cycles, the user attempts to increase their lock duration. veCVG and lockingPositionService will now be completely out of sync:

After 95 cycles the current time would be:

```
612,000 * 95 = 58,140,000
```

Whereas veCVG lock ended:

```
612,000 * 96 = 58,060,800
```

According to veCVG the position was unlocked at 58,060,800 and therefore increasing the lock time will revert due to L367

The result is another DOS that will cause the user loss of yield. During this time the user would also be excluded from taking place in any votes since their veCVG lock is expired.

Impact

Unlock DOS that cause loss of yield to the user

Code Snippet

CvgRewards.sol#L341-L349

Tool used

Manual Review

Recommendation

I would recommend against using block.timestamp for CVG cycles, instead using an absolute measurement like veCVG uses.

Discussion

walk-on-me

Hello dear judge,

this issue has been solved.

In order to solve it, we decided that when a user comes for locking or increase it's lock, we'll not take anymore the one in the `CvgControlTower`. If an user decide to performs a lock action between the timestamp where the cycle is updated and the CVG distribution (by the `CvgRewards`). He'll just lock for the next week, having no incidence on protocol.

Remarks : We are keeping the `cvgCycle` of the `CvgControlTower` for :

- Burning the NFT
- Claim the rewards of the Ys

You can check how by following the comments done here :

https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457499501 https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457504596 https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457504988 https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457505267 https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457508720
https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457510511

IAm0x52

Fix looks good. All cycles are now aligned by a computed weekly timestamp instead of using cvgControlTower

Issue M-5: SdtRewardReceiver#_withdrawRewards has incorrect slippage protection and withdraws can be sandwiched

Source:
https://github.com/sherlock-audit/2023-11-convergence-judging/issues/180

Found by

0x52, 0xkaden, Bauer, CL001, FarmerRick, caventa, cducrest-brainbot, detectiveking, hash, lemonmon, r0ck3tz

Summary

The _min_dy parameter of poolCvgSDT.exchange is set via the poolCvgSDT.get_dy method. The problem with this is that get_dy is a relative output that is executed at runtime. This means that no matter the state of the pool, this slippage check will never work.

Vulnerability Detail

SdtRewardReceiver.sol#L229-L236

```
if (isMint) {
    /// @dev Mint cvgSdt 1:1 via CvgToke contract
    cvgSdt.mint(receiver, rewardAmount);
} else {
    ICrvPoolPlain _poolCvgSDT = poolCvgSDT;
    /// @dev Only swap if the returned amount in CvgSdt is gretear than the
     ↪  amount rewarded in SDT
    _poolCvgSDT.exchange(0, 1, rewardAmount, _poolCvgSDT.get_dy(0, 1,
     ↪  rewardAmount), receiver);
}
```

When swapping from SDT to cvgSDT, get_dy is used to set _min_dy inside exchange. The issue is that get_dy is the CURRENT amount that would be received when swapping as shown below:

```
@view
@external
def get_dy(i: int128, j: int128, dx: uint256) -> uint256:
    """
    @notice Calculate the current output dy given input dx
    @dev Index values can be found via the `coins` public getter method
    @param i Index value for the coin to send
    @param j Index valie of the coin to recieve
    @param dx Amount of `i` being exchanged
    @return Amount of `j` predicted
    """
    rates: uint256[N_COINS] = self.rate_multipliers
    xp: uint256[N_COINS] = self._xp_mem(rates, self.balances)

    x: uint256 = xp[i] + (dx * rates[i] / PRECISION)
    y: uint256 = self.get_y(i, j, x, xp, 0, 0)
    dy: uint256 = xp[j] - y - 1
    fee: uint256 = self.fee * dy / FEE_DENOMINATOR
    return (dy - fee) * PRECISION / rates[j]
```

The return value is EXACTLY the result of a regular swap, which is where the problem is. There is no way that the exchange call can ever revert. Assume the user is swapping because the current exchange ratio is 1:1.5. Now assume their withdraw is sandwich attacked. The ratio is change to 1:0.5 which is much lower than expected. When get_dy is called it will simulate the swap and return a ratio of 1:0.5. This in turn doesn't protect the user at all and their swap will execute at the poor price.

Impact

SDT rewards will be sandwiched and can lose the entire balance

Code Snippet

SdtRewardReceiver.sol#L213-L245

Tool used

Manual Review

Recommendation

Allow the user to set _min_dy directly so they can guarantee they get the amount they want

Discussion

shalbe-cvg

Hello,

Thanks a lot for your attention.

After an in-depth review, we have to consider your issue as Confirmed. Not only users can get sandwiched but in most cases this exchange directly on the pool level would rarely succeed as `get_dy` returns the exact amount the user could get. We will add a slippage that users will setup.

Regards, Convergence Team

walk-on-me

This issue has been solved here :

https://github.com/Cvg-Finance/sherlock-cvg/pull/4

Follow the comment : https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457486906 https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457489632

IAm0x52

Fix looks good. User can now specify a min out parameter

Issue M-6: Division difference can result in a revert when claiming treasury yield and excess rewards to some users

Source:
https://github.com/sherlock-audit/2023-11-convergence-judging/issues/190

Found by

cergyk, hash, vvv

Summary

Different ordering of calculations are used to compute `ysTotal` in different situations. This causes the totalShares tracked to be less than the claimable amount of shares

Vulnerability Detail

`ysTotal` is calculated differently when adding to `totalSuppliesTracking` and when computing `balanceOfYsCvgAt`. When adding to `totalSuppliesTracking`, the calculation of `ysTotal` is as follows:

```
uint256 cvgLockAmount = (amount * ysPercentage) / MAX_PERCENTAGE;
uint256 ysTotal = (lockDuration * cvgLockAmount) / MAX_LOCK;
```

In `balanceOfYsCvgAt`, `ysTotal` is calculated as follows

```
uint256 ysTotal = (((endCycle - startCycle) * amount * ysPercentage) /
 ↪  MAX_PERCENTAGE) / MAX_LOCK;
```

This difference allows the `balanceOfYsCvgAt` to be greater than what is added to `totalSuppliesTracking`

POC

```
startCycle 357
endCycle 420
lockDuration 63
amount 2
ysPercentage 80
```

Calculation in `totalSuppliesTracking` gives:

```
uint256 cvgLockAmount = (2 * 80) / 100; == 1
uint256 ysTotal = (63 * 1) / 96; == 0
```

Calculation in `balanceOfYsCvgAt` gives:

```
uint256 ysTotal = ((63 * 2 * 80) / 100) / 96; == 10080 / 100 / 96 == 1
```

Example Scenario

Alice, Bob and Jake locks cvg for 1 TDE and obtains rounded up `balanceOfYsCvgAt`. A user who is aware of this issue can exploit this issue further by using `increaseLockAmount` with small amount values by which the total difference difference b/w the user's calculated `balanceOfYsCvgAt` and the accounted amount in `totalSuppliesTracking` can be increased. Bob and Jake claims the reward at the end of reward cycle. When Alice attempts to claim rewards, it reverts since there is not enough reward to be sent.

Impact

This breaks the shares accounting of the treasury rewards. Some user's will get more than the actual intended rewards while the last withdrawals will result in a revert

Code Snippet

`totalSuppliesTracking` calculation

In `mintPosition` https://github.com/sherlock-audit/2023-11-convergence/blob/main/sherlock-cvg/contracts/Locking/LockingPositionService.sol#L261-L263

In `increaseLockAmount` https://github.com/sherlock-audit/2023-11-convergence/blob/e894be3e36614a385cf409dc7e278d5b8f16d6f2/sherlock-cvg/contracts/Locking/LockingPositionService.sol#L339-L345

In `increaseLockTimeAndAmount` https://github.com/sherlock-audit/2023-11-convergence/blob/main/sherlock-cvg/contracts/Locking/LockingPositionService.sol#L465-L470

`_ysCvgCheckpoint` https://github.com/sherlock-audit/2023-11-convergence/blob/main/sherlock-cvg/contracts/Locking/LockingPositionService.sol#L577-L584

`balanceOfYsCvgAt` calculation https://github.com/sherlock-audit/2023-11-convergence/blob/main/sherlock-cvg/contracts/Locking/LockingPositionService.sol#L673-L675

Tool used

Manual Review

Recommendation

Perform the same calculation in both places

```
+++                     uint256 _ysTotal = (_extension.endCycle -
 ↪  _extension.cycleId)* ((_extension.cvgLocked *
 ↪  _lockingPosition.ysPercentage) / MAX_PERCENTAGE) / MAX_LOCK;
---     uint256 ysTotal = (((endCycle - startCycle) * amount *
 ↪  ysPercentage) / MAX_PERCENTAGE) / MAX_LOCK;
```

Discussion

walk-on-me

Hello

Indeed this is a real problem due the way that the invariant : *Sum of all balanceOfYsCvg > totalSupply*

And so some positions will become not claimable on the `YsDistributor`.

We'll correct this by computing the same way the ysTotal & ysPartial on the balanceYs & ysCheckpoint

Very nice finding, it'd break the claim for the last users to claim.

deadrosesxyz

Escalate The amounts are scaled up by 1e18. The rounding down problem comes when dividing by `MAX_PERCENTAGE` which equals 100. Worst case scenario (which will only happen if a user deposits an amount which is not divisible by 100), there will be rounding down of up to 99 wei. Not only it is insignificant, it is unlikely to happen as it requires a deposit of an amount not divisible by 1e2. Believe issue should be marked as low.

sherlock-admin2

> Escalate The amounts are scaled up by 1e18. The rounding down problem comes when dividing by `MAX_PERCENTAGE` which equals 100. Worst case scenario (which will only happen if a user deposits an amount which is not divisible by 100), there will be rounding down of up to 99 wei. Not only it is insignificant, it is unlikely to happen as it requires a deposit of an amount not divisible by 1e2. Believe issue should be marked as low.

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.

10xhash

(a) The rounding down is significant because it disallows the last claimers of a TDE cycle from obtaining their reward.

(b) An attacker can perform the attack which requires no expectations from the other users.

(c) The reasoning to classify non 1e2 amounts as unlikely would be the neatness part and the UI interacting users. There is a functionality provided by the CvgUtilities contract itself to lock Cvg using swap and bond tokens which shouldn't be discriminating non 1e2 amounts.

deadrosesxyz

Worst case scenario, only the last user will be unable to claim their rewards (even though I described above why it is highly unlikely). In the rare situation it happens, it can be fixed by simply sending a few wei to the contract.

nevillehuang

Imo, just off point 1 alone, this warrants medium severity at the least. The fact that a donation is required to fix this means there is a bug, and is not intended functionality of the function.

Czar102

I agree that this is a borderline low/med. I don't see a reason to discriminate nonzero deposits mod 100. That said, I am siding with the escalation – the loss here is insufficient to consider it a material loss of funds (at no point in the lifetime of the codebase will it surpass $1), and a loss of protocol functionality isn't serious enough if simply sending some dust to the contract will resolve the issue.

Planning to accept the escalation and consider this a low severity issue.

CergyK

@Czar102 please consider report #132 which I submitted which allows to steal an arbitrary amount from the rewards under some conditions, which is a higher impact.

My issue shares the same root cause as this one, so I did not escalate for deduplication. However if you think that this issue should be low, maybe it would be more fair to make my issue unique since the impact is sufficient.

nevillehuang

#132 and this #190 shares the same impact, if this is invalid, #132 should be invalid as well. Namely the following two impact:

(a) Last user withdrawals can revert

(b) Some users will gain more rewards at the expense of others.

Both examples present used involve relatively low amounts, so I'm unsure what is the exact impact

Comparing this issue attack path

> by using increaseLockAmount with small amount values by which the total difference difference b/w the user's calculated balanceOfYsCvgAt and the accounted amount in totalSuppliesTracking can be increased

and #132

> -> Alice locks some small amount for lockDuration = 64 so that it increases totalSupply by exactly 1 -> Alice proceeds to lock X times using the values:

Comparing this issue impact

> This breaks the shares accounting of the treasury rewards. Some user's will get more than the actual intended rewards while the last withdrawals will result in a revert

and #132

> Under specific circumstances (if the attacker is the only one to have allocated to YS during a TDE), an attacker is able to claim arbitrarily more rewards than is due to him, stealing rewards from other participants in the protocol

My opinion is both issues should remain valid medium severity issue based on impact highlighted in both issues.

CergyK

After some discussion with @nevillehuang, agree that issues should stay duplicated and valid `high/medium` given following reasons:

- Highest impact is: loss of arbitrary amount of present/future rewards (see #132 for explanation)
- Necessary condition of very low YS allocation is unlikely but not impossible since YS is not central in Convergence (YS allocation could be empty and the system would be working as expected)

deadrosesxyz

To summarize:

- Almost certainly in regular conditions, there will be no issue for any user whatsover.
- In some rare case, (see original escalation) there could be a small overdistribution of rewards (matter of a few wei). In the rarer case all users claim their rewards, the last unstaker will be unable to do so due to lack of funds. This is even more unlikely considering any time a user claims rewards, the amount they claim is rounded down, (due to built-in round down in solidity) leading to making the overdistribution smaller/inexistent. But even if all the conditions are met, the issue can be fixed by simply sending a few wei to the contract.
- The highest impact described in #132 requires for the total balance to not simply be *very low* but in fact to be just a few wei. Not only it has to be a few wei, but it has to be a few wei for at least 12 weeks (until TDE payout). It is absolutely unrealistic to have a few wei total balance for 12 weeks.

Issue should remain Low severity

10xhash

I agree that the fix of sending minor amounts of all reward tokens won't cost the team any considerable loss financially. But apart from the fix, the impact under reasonable conditions of user not being able to withdraw their rewards is certainly a major one. If issues existing on the contract are judged based on the ease of fixing/preventing, I think a lot more issues would exist under this category. Wouldn't it make almost all functionality breaks in up-gradable contracts low severity due to the fix being an upgrade?

Czar102

Due to the additional impact noted (thank you @CergyK) I think the loss can be sufficient to warrant a medium severity for this issue (loss of funds, but improbable assumptions are made).

Czar102

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- deadrosesxyz: accepted

walk-on-me

This issue has been solved here :

https://github.com/Cvg-Finance/sherlock-cvg/pull/4

Follow the comments :

- https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457453181
- https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457453459
- https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457453906
- https://github.com/Cvg-Finance/sherlock-cvg/pull/4#discussion_r1457455387

IAm0x52

Fix looks good. Order of operations has been updated to consistently reflect the proper value

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.