



**SHERLOCK**

# **SHERLOCK SECURITY REVIEW FOR**



**Prepared for:**

**Merit Circle**

**Prepared by:**

**Sherlock**

**Lead Security Expert:**

**WATCHPUG**

**Dates Audited:**

**October 10 - October 13, 2022**

**Prepared on:**

**November 7, 2022**

## Introduction

Merit Circle is creating a decentralized autonomous organization (DAO) that develops opportunities to earn through play for people who want to help build the metaverse.

## Scope

Contracts inside the contract folder in the Merit Liquidity Mining @ ce5feaa repo are in scope.

## 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
5	3

## Issues not fixed or acknowledged

Medium	High
0	0

## Security experts who found valid issues

WATCHPUG  
hyh  
bin2chen  
Jeiwan  
Lambda  
Ch\_301

rvierdiiev  
saian  
hickuphh3  
HonorLt  
Zarf  
CodingNameKiki

berndartmueller  
minhquanym  
ElKu  
ctf\_sec



## Issue H-1: Extend lock period should never result in a decrease of overall rewards ( $\text{total length of locked period} \times \text{shares}$ )

Source: <https://github.com/sherlock-audit/2022-10-merit-circle-judging/issues/109>

### Found by

WATCHPUG

### Summary

The current implementation may result in a burn of shares when `extendLock()`, which may result in a decrease of overall rewards ( $\text{total length of locked period} \times \text{shares}$ ).

### Vulnerability Detail

In the current implementation, when the user calls `extendLock()` for a deposit with duration1 at time  $t$ , with `_increaseDuration=duration2`, it will create a new lock with a lock period of  $\text{duration1} + \text{duration2} - \text{time}$ .

For example:

If the multiplier for a 6 mos lock is 150% and 200% for a 1 year lock.

- Alice deposited 100 \$MC tokens with the initial lock of 1 year, received 200 shares;
- 6 mos later, Alice called `extendLock()` to extend the deposit's lock for 10 more minutes (`MIN_LOCK_DURATION`), ~50 shares will be burned.

Expected result:

The total rewards should be no less than a 1 year lock;

Actual result:

The total rewards should be only  $200 \times 6\text{mos} + 150 \times 6\text{mos}$  vs  $200 \times 12\text{mos}$  for a 1 year lock;

By extend the lock for 10 more mins, Alice actually reduced the total rewards by 12.5%.

### Impact

Users may lose rewards by extending the lock.



## Code Snippet

<https://github.com/Merit-Circle/merit-liquidity-mining/blob/ce5feaae19126079d309ac8dd9a81372648437f1/contracts/TimeLockPool.sol#L148-L184>

## Tool used

Manual Review

## Recommendation

Consider introducing a new concept called `deferredLock` when `extendLock()` will result in the `newEndTime-block.timestamp < currentDuration`, in which case, instead of burning shares, it will set a deferred lock to be executable only after the current lock expires:

```
function extendLock(uint256 _depositId, uint256 _increaseDuration) external {
    // Check if actually increasing
    if (_increaseDuration == 0) {
        revert ZeroDurationError();
    }

    Deposit memory userDeposit = depositsOf[_msgSender()][_depositId];

    // Only can extend if it has not expired
    if (block.timestamp >= userDeposit.end) {
        revert DepositExpiredError();
    }

    // Enforce min increase to prevent flash loan or MEV transaction ordering
    uint256 increaseDuration = _increaseDuration.max(MIN_LOCK_DURATION);

    // New duration is the time expiration plus the increase
    uint256 duration = maxLockDuration.min(uint256(userDeposit.end -
    ↪ block.timestamp) + increaseDuration);

    uint256 mintAmount = userDeposit.amount * getMultiplier(duration) / 1e18;

    // Multiplier curve changes with time, need to check if the mint amount is
    ↪ bigger, equal or smaller than the already minted

    // If the new amount is bigger mint the difference
    if (mintAmount > userDeposit.shareAmount) {
        depositsOf[_msgSender()][_depositId].shareAmount = mintAmount;
        _mint(_msgSender(), mintAmount - userDeposit.shareAmount);

        depositsOf[_msgSender()][_depositId].start = uint64(block.timestamp);
    }
}
```



```

        depositsOf[_msgSender()][_depositId].end = uint64(block.timestamp) +
↪ uint64(duration);
        emit LockExtended(_depositId, _increaseDuration, _msgSender());

        // reset deferredLock if any
        if (deferredLocks[_msgSender()][_depositId] > 0) {
            deferredLocks[_msgSender()][_depositId] = 0;
        }
        // If the new amount is less then set a deferred lock
    } else if (mintAmount < userDeposit.shareAmount) {
        deferredLocks[_msgSender()][_depositId] = increaseDuration
    }
}

```

For expired locks, when there is a deferredLock, extend the lock and burn the differences in shares:

```

function kick(uint256 _depositId, address _user) external {
    if (_depositId >= depositsOf[_user].length) {
        revert NonExistingDepositError();
    }
    Deposit memory userDeposit = depositsOf[_user][_depositId];
    if (block.timestamp < userDeposit.end) {
        revert TooSoonError();
    }

    uint256 newSharesAmount = userDeposit.amount;

    uint256 deferredLock = deferredLocks[_user][_depositId];
    if (deferredLock != 0) {
        newSharesAmount = userDeposit.amount * getMultiplier(deferredLock) /
↪ 1e18;

        depositsOf[_user][_depositId].start = uint64(block.timestamp);
        depositsOf[_user][_depositId].end = uint64(block.timestamp) +
↪ uint64(deferredLock);
        emit LockExtended(_depositId, _increaseDuration, _user);
    }

    // burn pool shares
    _burn(_user, userDeposit.shareAmount - newSharesAmount);
}

```

## Discussion

federava



After some internal discussion we think it is better to revert if `extendLock` would result in burn of the tokens.

**federava**

PR from this issue

**jack-the-pug**

Fix confirmed



## Issue H-2: `increaseLock()` should read `userDeposit[_receiver]` instead of `depositsOf[_msgSender()]`

Source: <https://github.com/sherlock-audit/2022-10-merit-circle-judging/issues/102>

### Found by

WATCHPUG, saian, CodingNameKiki, Zarf, HonorLt, rvierdiev, hyh

### Summary

`TimeLockPool.sol` L203 should read `depositsOf[_receiver][_depositId]` as `userDeposit`.

### Vulnerability Detail

`TimeLockPool.sol` L230 in `increaseLock()` is loading `depositsOf[_msgSender()][_depositId]` as `userDeposit`, which will later be used to check if the deposit has expired (L206-208) and calculating the `remainingDuration` (L213).

This `remainingDuration` will be used to calculate the multiplier for the `mintAmount` at L215.

### Impact

As a result, the `_receiver` can receive a much larger shares.

For example, if the receiver only has 10 mins left in `depositsOf[_receiver][_depositId]`, but `depositsOf[_msgSender()][_depositId]` have 4 years left. The `mintAmount` will be 5x than expected.

Or fewer shares than expected when the caller's deposit's `remainingDuration` is shorter than the receiver's.

### Code Snippet

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/TimeLockPool.sol#L197-L222>

### Tool used

Manual Review

### Recommendation

Change L203 to:



```
Deposit memory userDeposit = depositsOf[_receiver][_depositId];
```

## Discussion

**federava**

Agree on the recommendation, thanks!

**federava**

PR from this issue

**jack-the-pug**

Fix confirmed





## Issue H-3: Unit isn't recalculated on curve modification with setCurvePoint

Source: <https://github.com/sherlock-audit/2022-10-merit-circle-judging/issues/101>

### Found by

bin2chen, Lambda, Ch\_301, Jeiwan, hyh

### Summary

TimeLockPool's setCurvePoint() can add or remove curve points, changing its length. It leaves `unit` variable that governs the duration to multiplier correspondence incorrect as it isn't updated.

### Vulnerability Detail

`unit` becomes incorrect after `onlyGov` setCurvePoint() updates the curve whenever its size changes.

As setCurvePoint() is an independent operation, the `unit` will just stay incorrect after it. I.e. setCurve() and \_\_TimeLockPool\_init() that update the `unit` are alternatives to setCurvePoint(), there is no use cases when they run after setCurvePoint(), fixing the variable. So here it is one scenario, where setCurvePoint() is run and `unit` is incorrect after that.

### Impact

When `unit` is incorrect the getMultiplier() calculations become incorrect too.

Suppose `curve.length` was 6, `unit=maxLockDuration/(curve.length-1)=maxLockDuration/5`, then setCurvePoint() shortened the curve, so `curve.length` becomes 3, but still `unit=maxLockDuration/5`. Now `getMultiplier(2*maxLockDuration/5)` yields maximum shares multiplier as the very end of the curve will be used, while the lock is only 2/5 of the `maxLockDuration`.

The impact is duration multiplier logic either removes the shares from users (when setCurvePoint() increased the length) or provides them with extra shares (when setCurvePoint() decreased the length), in violation of the desired logic. As the shares have monetary value and the precondition is just running a routine setup operation, setting the severity to be high.

### Code Snippet

setCurvePoint() can change curve length, but leaves `unit` variable intact:



<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/TimeLockPool.sol#L313-L337>

```
/**
 * @notice Can set a point of the curve.
 * @dev This function can replace any point in the curve by inputting the existing
 ↪ index,
 * add a point to the curve by using the index that equals the amount of points
 ↪ of the curve,
 * and remove the last point of the curve if an index greater than the length is
 ↪ used. The first
 * point of the curve index is zero.
 * @param _newPoint uint256 point to be set.
 * @param _position uint256 position of the array to be set (zero-based indexing
 ↪ convention).
 */
function setCurvePoint(uint256 _newPoint, uint256 _position) external onlyGov {
    if (_newPoint > maxBonus) {
        revert MaxBonusError();
    }
    if (_position < curve.length) {
        curve[_position] = _newPoint;
    } else if (_position == curve.length) {
        curve.push(_newPoint);
    } else {
        if (curve.length - 1 < 2) {
            revert ShortCurveError();
        }
        curve.pop();
    }
    emit CurveChanged(_msgSender());
}
```

Currently unit is recalculated on initialization:

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/TimeLockPool.sol#L62>

```
unit = _maxLockDuration / (curve.length - 1);
```

And on altering curve length with setCurve(), which rebuilds the entire curve:

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/TimeLockPool.sol#L284-L309>

```
// same length curves
if (curve.length == _curve.length) {
    for (uint i=0; i < curve.length; i++) {
```



```

        curve[i] = maxBonusError(_curve[i]);
    }
    // replacing with a shorter curve
} else if (curve.length > _curve.length) {
    for (uint i=0; i < _curve.length; i++) {
        curve[i] = maxBonusError(_curve[i]);
    }
    uint initialLength = curve.length;
    for (uint j=0; j < initialLength - _curve.length; j++) {
        curve.pop();
    }
    unit = maxLockDuration / (curve.length - 1);
    // replacing with a longer curve
} else {
    for (uint i=0; i < curve.length; i++) {
        curve[i] = maxBonusError(_curve[i]);
    }
    uint initialLength = curve.length;
    for (uint j=0; j < _curve.length - initialLength; j++) {
        curve.push(maxBonusError(_curve[initialLength + j]));
    }
    unit = maxLockDuration / (curve.length - 1);
}

```

unit maps \_lockDuration to the point on the curve, breaking this link if curve length had changed, while unit stayed the same:

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/TimeLockPool.sol#L233-L246>

```

function getMultiplier(uint256 _lockDuration) public view returns(uint256) {
    // There is no need to check _lockDuration amount, it is always checked
    ↪ before
    // in the functions that call this function

    // n is the time unit where the lockDuration stands
    uint n = _lockDuration / unit;
    // if last point no need to interpolate
    // trim de curve if it exceeds the maxBonus // TODO check if this is needed
    if (n == curve.length - 1) {
        return 1e18 + curve[n];
    }
    // linear interpolation between points
    return 1e18 + curve[n] + (_lockDuration - n * unit) * (curve[n + 1] -
    ↪ curve[n]) / unit;
}

```



## Tool used

Manual Review

## Recommendation

Consider recalculation `unit` on the spot each time:

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/TimeLockPool.sol#L313-L337>

```
/**
 * @notice Can set a point of the curve.
 * @dev This function can replace any point in the curve by inputing the
↪ existing index,
 * add a point to the curve by using the index that equals the amount of
↪ points of the curve,
 * and remove the last point of the curve if an index greated than the length
↪ is used. The first
 * point of the curve index is zero.
 * @param _newPoint uint256 point to be set.
 * @param _position uint256 position of the array to be set (zero-based
↪ indexing convention).
 */
function setCurvePoint(uint256 _newPoint, uint256 _position) external onlyGov
↪ {
    if (_newPoint > maxBonus) {
        revert MaxBonusError();
    }
    if (_position < curve.length) {
        curve[_position] = _newPoint;
    } else if (_position == curve.length) {
        curve.push(_newPoint);
+        unit = maxLockDuration / (curve.length - 1);
    } else {
        if (curve.length - 1 < 2) {
            revert ShortCurveError();
        }
        curve.pop();
+        unit = maxLockDuration / (curve.length - 1);
    }
    emit CurveChanged(_msgSender());
}
```

`setCurvePoint()` is rare enough operation and cost of incorrect `unit` is rather big here, so the additional gas cost is justified.



## Discussion

**federava**

Agree on the recommendation, thanks!

**federava**

PR from this issue

**jack-the-pug**

Fix confirmed



## Issue M-1: Curve points should be guaranteed to be monotonic increasing

Source: <https://github.com/sherlock-audit/2022-10-merit-circle-judging/issues/111>

### Found by

berndartmueller, minhquanym, WATCHPUG, rvierdiev, hickuphh3, ElKu

### Summary

Lack of checks to ensure the points in the curve are monotonic increasing, which can result in a malfunction of `deposit()` / `extendLock()` due to underflow when the curve is not set properly.

### Vulnerability Detail

In the current implementation, `getMultiplier()` assume the later point in the curve is always bigger than the previous point, otherwise `curve[n+1]-curve[n]` will revert due to underflow.

However, since there is no check in `__TimeLockPool_init()` / `setCurve()` / `setCurvePoint()` to guarantee that, a lower point can actually be set after a higher point.

### Impact

`deposit()` / `extendLock()` may revert due to underflow.

### Code Snippet

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/TimeLockPool.sol#L35-L63>

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/TimeLockPool.sol#L280-L311>

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/TimeLockPool.sol#L322-L337>

### Tool used

Manual Review



## Recommendation

Consider adding a new internal function to validate the curve points:

```
function checkCurve(uint256[] calldata _curve) internal {
    if (_curve.length < 2) {
        revert ShortCurveError();
    }
    for (uint256 i; i < _curve.length - 1; ++i) {
        if (
            _curve[i + 1] < _curve[i]
        ) {
            revert CurveIncreaseError();
        }
    }
}
```

## Discussion

**federava**

Agree on the recommendation. Using that logic everywhere the curve is set should be sufficient: `setCurve()`, `setCurvePoint()` and in `__TimeLockPool_init()`. Thanks!

**federava**

PR from this issue

**jack-the-pug**

Fix confirmed



## Issue M-2: Expired locks should not continue to earn rewards at the original high multiplier rate

Source: <https://github.com/sherlock-audit/2022-10-merit-circle-judging/issues/108>

### Found by

WATCHPUG

### Summary

Expired locks should be considered as same as the deposits with no lock.

### Vulnerability Detail

The current implementation allows the deposits with expired locks to continue to enjoy the original high multiplier rate, while they can withdraw anytime they want.

The multiplier of shares amount is essentially a higher reward rate (APR) for longer period of locks.

For example:

If the regular APR is 2%; Locking for 4 years will boost the APR to 10%.

- Alice deposited 1M \$MC tokens and got 10% APR;
- 4 years later, Alice's deposit's lock was expired.

Expected result:

The new APR for Alice's deposit is 2%;

Actual result:

Alice can continue to enjoy a 10% APR while she can withdraw anytime.

### Impact

Users with expired locks will take more rewards than expected, which means fewer rewards for other users.

### Code Snippet

<https://github.com/Merit-Circle/merit-liquidity-mining/blob/ce5feaae19126079d309ac8dd9a81372648437f1/contracts/TimeLockPool.sol#L116-L135>





## Tool used

Manual Review

## Recommendation

Curve's Gauge system introduced a method called `kick()` which allows the expired (zeroed) veCRV users to be kicked from the rewards.

See: <https://github.com/curvefi/curve-dao-contracts/blob/master/contracts/gauges/LiquidityGaugeV5.vy#L430-L446>

A similar method can be added to solve this issue:

```
function kick(uint256 _depositId, address _user) external {
    if (_depositId >= depositsOf[_user].length) {
        revert NonExistingDepositError();
    }
    Deposit memory userDeposit = depositsOf[_user][_depositId];
    if (block.timestamp < userDeposit.end) {
        revert TooSoonError();
    }

    // burn pool shares
    _burn(_user, userDeposit.shareAmount - userDeposit.amount);
}
```

## Discussion

**federava**

Agree on the recommendation, will implement kick function. Noticing that shares go back to a 1:1 ratio and that the function can be called by anyone is a good design choice.

**federava**

PR from this issue

**jack-the-pug**

Fix confirmed



## Issue M-3: `escrowedReward` will be frozen in the contract if `escrowPool==address(0)` but `escrowPortion>0`

Source: <https://github.com/sherlock-audit/2022-10-merit-circle-judging/issues/107>

### Found by

bin2chen, berndartmueller, WATCHPUG, saian, hickuphh3, Jeiwan, ctf\_sec

### Summary

A portion of users' reward, which is expected to be "escrowed", will be frozen in the pool contract if `escrowPool==address(0)` but `escrowPortion>0`.

### Vulnerability Detail

Setting `_escrowPool` to `address(0)` is allowed in `__BasePool_init()`:

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/base/BasePool.sol#L75-L77>

However, when `escrowPortion>0`, if `escrowPool==address(0)`, `claimRewards()` will still only transferring `nonEscrowedRewardAmount` to the `_receiver` and left the `escrowedRewardAmount` in the contract.

### Impact

As a result, a portion (`escrowPortion`) of the rewards will be frozen in the contract, and there is no way for the users or even the admin to retrieve these rewards.

### Code Snippet

<https://github.com/Merit-Circle/merit-liquidity-mining/blob/ce5feaae19126079d309ac8dd9a81372648437f1/contracts/base/BasePool.sol#L100-L115>

### Tool used

Manual Review

### Recommendation

Change to:

```
function claimRewards(address _receiver) external {
    uint256 rewardAmount = _prepareCollect(_msgSender());
    uint256 escrowedRewardAmount = 0;
```



```

    if(escrowPortion != 0 && address(escrowPool) != address(0)) {
        escrowedRewardAmount = rewardAmount * escrowPortion / 1e18;
        if (escrowedRewardAmount != 0) {
            escrowPool.deposit(escrowedRewardAmount, escrowDuration, _receiver);
            rewardAmount -= escrowedRewardAmount;
        }
    }

    // ignore dust
    if(rewardAmount > 1) {
        rewardToken.safeTransfer(_receiver, rewardAmount);
    }

    emit RewardsClaimed(_msgSender(), _receiver, escrowedRewardAmount,
↳   rewardAmount);
}

```

## Discussion

### **federava**

Conditions stablished in this issue, that is, `escrowPool==address(0)` and `escrowPortion>0` will never be met because pools are deployed with:

`escrowPool!=address(0)` and `escrowPortion>0` or `escrowPool==address(0)` and `escrowPortion>0`

As a priority we could check on deployment that such conditions are not met.

### **federava**

PR from this issue

### **jack-the-pug**

Fix confirmed



## Issue M-4: Front run `distributeRewards()` can steal the newly added rewards

Source: <https://github.com/sherlock-audit/2022-10-merit-circle-judging/issues/106>

### Found by

WATCHPUG, hickuphh3, hyh

### Summary

A surge of `pointsPerShare` on each `distributeRewards()` call can be used by the attacker to steal part of the newly added rewards.

### Vulnerability Detail

Every time the `distributeRewards()` gets called, there will be a surge of `pointsPerShare` for the existing stakeholders.

This enables a well-known attack vector, in which the attacker will deposit a huge amount of underlying tokens and take a large portion of the pool, then trigger the surge, and exit right after.

### Impact

While the existence of the `MIN_LOCK_DURATION` prevented the usage of flashloan, it's still possible for the attackers with sufficient funds or can acquire sufficient funds in other ways.

In which case, the attack is quite practical and effectively steal the major part of the newly added rewards

### Code Snippet

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/base/BasePool.sol#L95-L98>

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/base/AbstractRewards.sol#L89-L99>

### Tool used

Manual Review



## Recommendation

Consider using a `rewardRate`-based gradual release model, pioneered by Synthetix's StakingRewards contract.

See: <https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol#L113-L132>

## Discussion

**federava**

Raising the MIN\_LOCK\_DURATION from 10 minutes to 1 day.

**federava**

PR from this issue

**jack-the-pug**

Fix confirmed



## Issue M-5: First user can inflate `pointsPerShare` and cause `_correctPoints()` to revert due to overflow

Source: <https://github.com/sherlock-audit/2022-10-merit-circle-judging/issues/103>

### Found by

bin2chen, WATCHPUG

### Summary

`pointsPerShare` can be manipulated by the first user and cause `_correctPoints()` to revert later.

### Vulnerability Detail

`POINTS_MULTIPLIER` is an unusually large number as a precision fix for `pointsPerShare`: `type(uint128).max = 3.4e38`.

This makes it possible for the first user to manipulate the `pointsPerShare` to near `type(int256).max` and a later regular user can trigger the overflow of `_shares*_shares` \* in `_correctPoints()`.

### PoC

1. `deposit(1wei)` lock for 10 mins, `mint()` 1 wei of shares;
2. `distributeRewards(1000e18), pointsPerShare+=1000e18*type(uint128).max/1 == 3e59;`
3. the victim `deposit(100e18)` for 1 year, `mint()` 150e18 shares;
4. `_shares*pointsPerShare== -150e18*3e59== -4.5e+79` which exceeds `type(int256).min`, thus the transaction will revert.

The attacker can also manipulate `pointsPerShare` to a slightly smaller number, so that `_shares*pointsPerShare` will only overflow after a certain amount of deposits.

### Impact

By manipulating the `pointPerShare` precisely, the attacker can make it possible for the system to run normally for a little while and only to explode after a certain amount of deposits, as the `pointsPerShare` will be too large by then and all the `_mint` and `_burn` will revert due to overflow in `_correctPoints()`.

The users who deposited before will be unable to withdraw their funds.



## Code Snippet

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/base/AbstractRewards.sol#L89-L99>

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/base/BasePool.sol#L80-L88>

<https://github.com/sherlock-audit/2022-10-merit-circle/blob/main/merit-liquidity-mining/contracts/base/AbstractRewards.sol#L125-L127>

## Tool used

Manual Review

## Recommendation

1. Add a minimal mint amount requirement for the first minter and send a portion of the initial shares to gov as a permanent reserve to avoid `pointPerShare` manipulating.
2. Use a smaller number for `POINTS_MULTIPLIER`, eg, `1e12`.

## Discussion

**federava**

Agree with the recommendation. Requiring a minimal amount for first minter/minting on deployment by governance should be enough to prevent the described manipulation.

**federava**

PR from this issue

**federava**

The approach for this fix will be creating the first deposit with `amount>1e18` on deployment.

Using a check as a caution only for the first deposit is not convenient as it translates into increasing gas for the users.

In this PR has the removal of the previously implemented code.

**jack-the-pug**

The gov address will be the first depositor to deposit `amount>1e18` on deployment. They will never remove their shares, therefore the `totalShares` will never be lower than `1e18`.

**federava**



Agree. That is how we are going to proceed.

