Protocol Audit Report

Version 1.0

Cyfrin.io

May 21, 2024

Protocol Audit Report

EPSec

May 20, 2024

Prepared by: EPSec

Lead Auditors:

- PeterSR
-
- 0x539.eth

Table of Contents

Protocol Summary

The locked staking contract allows users to lock an ERC-20 token at various intervals and multipliers and receive fee distributions according to their amounts staked and multipliers.

Disclaimer

The EPSec team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

StakingV2/src/Lock.sol

StakingV2/src/interfaces/ILock.sol

StakingV2/src/interfaces/ILockList.sol

StakingV2/src/libraries/AddressPagination.sol

StakingV2/src/libraries/LockList.solsol

Roles

Executive Summary

Issues found

2 High

4 Medium

2 Low

2 Informational

Findings

High

[H-1] WithdrawAllTokens_Broken_Pagination

Summary

The `withdrawAllUnlockedToken` function in the `Lock` contract exhibits flawed pagination implementation, potentially rendering the function unusable due to high gas consumption. Moreover, the pagination mechanism within the `getLocks` function of the `LockList` contract also suffers from inefficiencies, returning unnecessary empty records. These critical issues significantly impact the contracts' functionality and efficiency, necessitating immediate corrective action.

Vulnerability Detail

In `Lock:withdrawAllUnlockedToken`, pagination is incorrectly implemented, leading to excessive gas consumption. The current logic iterates over the entire array within each execution of the while loop, rendering pagination ineffective. `LockList::getLocks` utilizes `lockCount` instead of `limit` as a parameter, causing the function to iterate over the entire array unnecessarily, exacerbating gas inefficiency.

Impact

The flawed pagination implementation in both contracts severely hampers their usability and efficiency. The `withdrawAllUnlockedToken` function, considered a key functionality per documentation, risks becoming practically unusable due to high gas consumption. Additionally, the retrieval of unnecessary empty records in `getLocks` introduces inefficiencies, potentially hindering contract interactions and escalating transaction costs.

The severity of the vulnerability comes from the fact that both functions (`Lock::withdrawAllUnlockedToken` and `LockList::getLocks`) have problems with pagination. In combination the function is unusable due to high gas costs.

Code Snippet

```solidity
function withdrawAllUnlockedToken() external override nonReentrant {
    uint256 lockCount = locklist.lockCount(msg.sender); // Fetch the
        total number of locks for the caller.
    uint256 page;
    uint256 limit;
    uint256 totalUnlocked;
    while (limit < lockCount) {
        LockedBalance[] memory lockedBals = locklist.getLocks(msg.
            sender, page, lockCount); // Retrieves a page of locks for
            the user.
        for (uint256 i = 0; i < lockedBals.length; i++) {
            if (lockedBals[i].unlockTime != 0 && lockedBals[i].
                unlockTime < block.timestamp) {
                totalUnlocked += lockedBals[i].amount; // Adds up the
                    amount from all unlocked balances.
                locklist.removeFromList(msg.sender, lockedBals[i].
                    lockId); // Removes the lock from the list.
            }
        }

        limit += 10; // Moves to the next page of locks.
        page++;
    }

    IERC20(stakingToken).safeTransfer(msg.sender, totalUnlocked); //
        Transfers the total unlocked amount to the user.
    emit WithdrawAllUnlocked(msg.sender, totalUnlocked); // Emits an
        event logging the withdrawal.
}
```

Recommendation

Consider directly iterating on all locks just once, without using while cycle as shown below:

```solidity
function withdrawAllUnlockedToken() external override nonReentrant {
    uint256 lockCount = locklist.lockCount(msg.sender); // Fetch the
        total number of locks for the caller.

-   uint256 page;
-   uint256 limit;
```

```
    uint256 totalUnlocked;

+   LockedBalance[] memory lockedBals = locklist.getLocks(
+       msg.sender,
+       0,
+       lockCount
+   );

+   for (uint256 i = 0; i < lockedBals.length; i++) {
+       if (
+           lockedBals[i].unlockTime != 0 &&
+           lockedBals[i].unlockTime < block.timestamp
+       ) {
+       totalUnlocked += lockedBals[i].amount; // Adds up the amount from all unlocked balances.
+       locklist.removeFromList(msg.sender, lockedBals[i].lockId); // Removes the lock from the list.
+       }
+   }

-   while (limit < lockCount) {
-       LockedBalance[] memory lockedBals = locklist.getLocks(msg.sender, page, lockCount); // Retrieves a page of locks for the user.
-       for (uint256 i = 0; i < lockedBals.length; i++) {
-           if (lockedBals[i].unlockTime != 0 && lockedBals[i].unlockTime < block.timestamp) {
-               totalUnlocked += lockedBals[i].amount; // Adds up the amount from all unlocked balances.
-               locklist.removeFromList(msg.sender, lockedBals[i].lockId); // Removes the lock from the list.
-           }
-       }
-       limit += 10; // Moves to the next page of locks.
-       page++;
-   }
    IERC20(stakingToken).safeTransfer(msg.sender, totalUnlocked); // Transfers the total unlocked amount to the user.

    emit WithdrawAllUnlocked(msg.sender, totalUnlocked); // Emits an event logging the withdrawal.
}
```

Proof of concept

1. Add this test to `Withdraw.t.sol`:

```
function testWithrdaw_StakedTokens() public {
    vm.prank(user1);
    lock.stake(100e18, user1, 0);
    uint256 i;

    for (i; i < 50; i++) {
        vm.prank(user1);
        lock.stake(1, user1, 0);
    }

    lock.withdrawAllUnlockedToken();
}
```

2. After running this command you can see the expected gas consumption `forge test --match-test testWithrdaw_StakedTokens -vvv --gas-report`
3. Then you can change the implementation to the proprosed one in the previous point `Recomendation` and run again the above command and see the consumed gas.

[H-2] WithdrawAllTokens_Broken_Pagination

Summary

The audit identified an issue in the smart contract where the order of function calls `exitLateById` and `notifyUnseenReward` affects the user's reward balance. Specifically, users do not receive their stake rewards if they exit late before unseen rewards are notified.

Vulnerability Detail

The issue lies in the handling of user exits and reward notifications. When a user calls `exitLateById` before `notifyUnseenReward`, their rewards are not updated correctly, leading to a loss of their due rewards. Conversely, if `notifyUnseenReward` is called before `exitLateById`, the rewards are processed as expected. This inconsistency indicates a flaw in how the contract manages reward calculations relative to user exits.

Impact

The impact of this vulnerability is that users who exit late before unseen rewards are notified will not receive their due rewards. This can lead to loss of funds for users and undermines the trust and reliability of the staking mechanism.

Code Snippet

Tests demonstrating the issue:

```
function testUserExitLateAfterNotifyUnseen() public {
    // Users stake tokens
    vm.prank(user1);
    lock.stake(100, user1, 0);
    vm.prank(user2);
    lock.stake(100, user2, 0);
    vm.prank(user3);
    lock.stake(100, user3, 0);
    vm.prank(deployer);
    _rewardToken.transfer(address(lock), 300);
    vm.prank(user2);
    LockedBalance[] memory locks = lockList.getLocks(user2, 0, 10);
    vm.prank(deployer);
    lock.notifyUnseenReward(rewardTokens);
    vm.prank(user2);
    lock.exitLateById(locks[0].lockId);

    vm.warp(block.timestamp + 11);
    vm.prank(user2);
    lock.getAllRewards();
    assertEq(_rewardToken.balanceOf(user2), 0);
}

function testUserExitLateBeforeNotifyUnseen() public {
    // Users stake tokens
    vm.prank(user1);
    lock.stake(100, user1, 0);
    vm.prank(user2);
    lock.stake(100, user2, 0);
    vm.prank(user3);
    lock.stake(100, user3, 0);
    vm.prank(deployer);
    _rewardToken.transfer(address(lock), 300);
    vm.prank(user2);
    LockedBalance[] memory locks = lockList.getLocks(user2, 0, 10);
    vm.prank(user2);
    lock.exitLateById(locks[0].lockId);
    vm.prank(deployer);
    lock.notifyUnseenReward(rewardTokens);

    vm.warp(block.timestamp + 11);
    vm.prank(user2);
    lock.getAllRewards();
    assertEq(_rewardToken.balanceOf(user2), 100);
}
```

Both tests fail. In the first one the balance is equal to 100 when we expect 0, in the second one the balance is equal to 0 when we expect 100. The only difference between the two tests is the order in which we are calling `Lock::notifyUnseenReward` and `Lock::exitLateById`.

Tool used

Manual Review

Recommendation

To fix this issue, ensure that rewards are consistently updated regardless of the order in which `Lock::exitLateById` and `Lock::notifyUnseenReward` are called. This can be done by modifying the reward update logic to correctly account for pending rewards whenever a user exits late or unseen rewards are notified.

Medium

[M-1] No Proper Validation For Removing Locks

Summary

The current implementation of `LockList::removeFromList` lacks proper validation, allowing the removal of locks belonging to other users.

Vulnerability Detail

The flaw lies in `LockList::removeFromList`, where the method `EnumerableSet::remove` fails to revert when attempting to remove a lock not owned by the user parameter. Instead, it returns false, leading to an inconsistent state. Consequently, if the method successfully removes the lock, `LockList::lockIndexesByUser` will contain a `lockId` which is already removed.

Impact

This vulnerability poses a significant risk as it enables the unauthorized removal of locks belonging to other users. Moreover, given the upgradability of `Lock.sol`, this vulnerability could be exploited in unforeseen ways, potentially leading to further system compromise.

Proof of concept

1. Add the following test to LockList:

```
function testRemove_lockWhichIsAssignedToAnotherUser() public {
        vm.startPrank(deployer);
        uint256 i;
        LockedBalance memory lockedBalance = LockedBalance({
            lockId: 0,
            amount: 100,
            unlockTime: i,
            multiplier: i + 1,
            lockTime: i + 2,
            lockPeriod: 0,
            exitedLate: false
        });
        LockedBalance memory lockedBalanceForUser2 = LockedBalance({
            lockId: 0,
            amount: 100,
            unlockTime: i,
            multiplier: i + 1,
            lockTime: i + 2,
            lockPeriod: 0,
            exitedLate: false
        });
        lockList.addToList(user1, lockedBalance);
        lockList.addToList(user2, lockedBalanceForUser2);
        vm.expectRevert();
        lockList.removeFromList(user2, 0);
        uint256 lockCount = lockList.lockCount(user2);
        assertEq(lockCount, 0);
    }
```

2. Run it with `forge test --match-test testRemove_lockWhichIsAssignedToAnotherUser`

Code Snippet

```
function removeFromList(
        address user,
        uint256 lockId
) public override onlyOwner {
    delete lockById[lockId];

    lockIndexesByUser[user].remove(lockId);
    emit LockRemoved(user, lockId);
}
```

Tool used

Manual Review

Recommendation

Implement a check in the `LockList::removeFromList` function to ensure that the lock being removed belongs to the specified user.

For example, this could be handled as follows:

```
function removeFromList(
        address user,
        uint256 lockId
    ) public override onlyOwner {

        delete lockById[lockId];
+       if(!lockIndexesByUser[user].remove(lockId)) {
+           revert RemoveWasNotSuccessful();
+       }
-       lockIndexesByUser[user].remove(lockId);
        emit LockRemoved(user, lockId);
    }
```

[M-2] Random Items From At

Summary

The `LockList::getLock` and `LockList::getLocks` functions in the smart contract may return random items due to the usage of `EnumerableSet::at`, which does not guarantee the order of elements. This can lead to unpredictable behaviour when retrieving locked balances for a user.

Vulnerability Detail

The `getLock` function retrieves a lock based on the user's address and an index:

```
function getLock(
    address user,
    uint256 index
) external view override returns (LockedBalance memory) {
    return lockById[lockIndexesByUser[user].at(index)];
}
```

The `getLocks` function retrieves locks based on the user's address:

```
function getLocks(
    address user,
    uint256 page,
    uint256 limit
) public view override returns (LockedBalance[] memory) {
    LockedBalance[] memory locks = new LockedBalance[](limit);
    uint256 lockIdsLength = lockIndexesByUser[user].length();

    uint256 i = page * limit;
    for (;i < (page + 1) * limit && i < lockIdsLength; i ++) {
        locks[i - page * limit]= lockById[lockIndexesByUser[user].at(i)
            ];
    }
    return locks;
}
```

The issue arises from the use of `EnumerableSet::at`, a function from the OpenZeppelin library, which does not ensure the order of elements. As specified in the OpenZeppelin documentation:

Note that there are no guarantees on the ordering of values inside the array, and it may change when more values are added or removed.

This means that the index provided to `getLock` may not correspond to the expected locked balance, leading to potential inconsistencies and errors.

Impact

The unpredictability of the `getLock` and `getLocks` functions' output can lead to incorrect locked balance retrievals for users. This can affect the accuracy of the contract's state and potentially disrupt user interactions that depend on specific orderings of locked balances.

After contacting with the protocol team it is known that `getLock` function is only used when interacting with the contract from a FrontEnd.

However, the `getLocks` function is used throughout the whole protocol providing important data and the likelihood of a vulnerability happening due to the usage of `EnumerableSet::at` there is high with a high impact.

Another major problem that could occur here is that if `LockList::getLock` returns wrong Lock in the frontend and using this wrong Lock to execute `Lock::earlyExitById` or `LockList::exitLateById` this will lead to incorrect exit.

Code Snippet

```
function getLock(
    address user,
    uint256 index
) external view override returns (LockedBalance memory) {
    return lockById[lockIndexesByUser[user].at(index)];
}
```

```
function getLocks(
    address user,
    uint256 page,
    uint256 limit
) public view override returns (LockedBalance[] memory) {
    LockedBalance[] memory locks = new LockedBalance[](limit);
    uint256 lockIdsLength = lockIndexesByUser[user].length();

    uint256 i = page * limit;
    for (;i < (page + 1) * limit && i < lockIdsLength; i ++) {
        locks[i - page * limit]= lockById[lockIndexesByUser[user].at(i)
            ];
    }
    return locks;
}
```

Tool used

Manual Review

Recommendation

Avoid using `EnumerableSet::at` if the order of elements is critical. Consider using a different data structure that guarantees the order of elements, or implement a mechanism to maintain and retrieve the correct order of locked balances.

[M-3] Unexpected results in paginations

Summary

The `AddressPagination::paginate` function returns incorrect results when the requested page exceeds the available data, resulting in unexpected behaviour such as trailing address(0) in the output array.

The same faulty logic can be observed in the `LockList::getLocks`

Vulnerability Detail

The current implementation of the `AddressPagination::paginate` function does not handle cases where the requested page and limit combination exceeds the bounds of the input array. Specifically, if the computed index exceeds the array length, the function inserts an address(0) into the results array. This behaviour can lead to silent errors and unexpected outputs, which may cause issues in the larger application logic.

The current implementation of the `LockList::getLocks` function does not handle cases where the requested page and limit combination exceeds the bounds of the input array. Specifically, if the computed index exceeds the array length, the function inserts an empty locked balance into the results array. This behaviour can lead to silent errors and unexpected outputs, which may cause issues in the larger application logic.

Impact

If the function is called with arguments that request data beyond the available array length, it returns a result array padded with zeros instead of reverting or indicating an error. This could lead to misleading data being processed in subsequent operations, potentially causing logical errors and undermining the reliability of the application.

While the `AddressPagination::paginate` function is currently not used anywhere, the `LockList::getLocks` function is used in `Lock::withdrawAllUnlockedToken` and `LockList::lockedBalances` functions (these are of high importance to the contract's behaviour).

Code Snippet

Original Functions:

```
function paginate(
    address[] memory array,
    uint256 page,
    uint256 limit
) internal pure returns (address[] memory result) {
    result = new address[](limit);
    for (uint256 i = 0; i < limit; i++) {
        if (page * limit + i >= array.length) {
            result[i] = address(0);
        } else {
            result[i] = array[page * limit + i];
        }
    }
}
```

```
function getLocks(
    address user,
    uint256 page,
    uint256 limit
)
public view override returns (LockedBalance[] memory) {
    LockedBalance[] memory locks = new LockedBalance[](limit);
    uint256 lockIdsLength = lockIndexesByUser[user].length();
    uint256 i = page * limit;
    for (;i < (page + 1) * limit && i < lockIdsLength; i ++) {
        locks[i - page * limit]= lockById[lockIndexesByUser[user].at(i)
            ];
    }
    return locks;
}
```

Proof Of Concept

Example of Incorrect Behaviour for AddressPagination::paginate:

Let's have a look with the same `AddressPagination::paginate` function but with integers.

The function looks like this:

```
function paginate(
    int256[] memory array,
    uint256 page,
    uint256 limit
) external pure returns (int256[] memory result) {
    result = new int256[](limit);
    for (uint256 i = 0; i < limit; i++) {
        if (page * limit + i >= array.length) {
            result[i] = 0;
        } else {
            result[i] = array[page * limit + i];
        }
    }
}
```

- Calling the function with `([1,2,3,4,5], 1, 3)` returns `[4, 5, 0]` instead of `[4, 5]`.
- Calling the function with `([1,2,3,4,5], 2, 3)` returns `[0, 0, 0]` instead of reverting or returning `[]`.

Example of Incorrect Behaviour for LockList::getLocks:

Let's write a test to see if the `lockCount` is the same as if we gather all locks using `LockList::getLocks` and take the array's length:

```
function testGetLocksPagination() public {
    vm.startPrank(deployer);
    uint256 i;
    for (i; i < 10; i++) {
        LockedBalance memory lockedBalance = LockedBalance({
            lockId: 0,
            amount: 100,
            unlockTime: i,
            multiplier: i + 1,
            lockTime: i + 2,
            lockPeriod: 0,
            exitedLate: false
        });

        lockList.addToList(user1, lockedBalance);
    }
    uint256 lockCount = lockList.lockCount(user1);
    uint256 lockCountWithPagination = lockList
        .getLocks(user1, 0, 11)
        .length;
    assertEq(lockCountWithPagination, lockCount);
    vm.stopPrank();
}
```

This test adds 10 locked balances to `user1`. After this it gets the lockCount using `LockList::lockCount` and tries to get 11 locks from page 0 using `LockList::getLocks`. The expected behaviour is for both lockCount and lockCountWithPagination to be equal to each other and to be equal to 10 as that's how much locked balances `user1` possesses.

However, the test fails due to the fact that `LockList::getLocks` will always return a list with the number of elements set to the limit variable (even if the user has no locked balances).

The test can be ran with `forge test --match-test testGetLocksPagination`.

Tool used

Manual Review

Recommendation

Refactor the functions to include bounds checking and ensure it reverts on invalid input, preventing the return of incorrect and misleading data.

Proposed Fix:

```
function paginate(
    address[] memory array,
    uint256 page,
    uint256 limit
) external pure returns (address[] memory result) {
+   require(array.length > page * limit, "Invalid request for pagination");
+   uint256 elements = limit;

+   if((page + 1) * limit >= array.length) {
+       elements -= (((page + 1) * limit) - array.length);
+   }

-   result = new address[](limit);
+   result = new address[](elements);

-   for (uint256 i = 0; i < limit; i++) {
+   for (uint256 i = 0; i < elements; i++) {
-       if (page * limit + i >= array.length) {
-           result[i] = address(0);
-       } else {
        result[i] = array[page * limit + i];
-       }
    }

+   return result;
}
```

This revised `AddressPagination::paginate` function ensures that any attempt to paginate beyond the available data will result in a revert, thus avoiding silent errors and ensuring consistent and reliable behaviour.

A similar rework should be done to `LockList::getLocks`.

[M-4] Wrong iteration length

Summary

The `Lock::notifyUnseenReward` function does not work as intended, leading to potential reverts and incorrect processing of reward tokens.

Vulnerability Detail

The function aims to check and update unseen rewards for a given list of reward tokens. However, it mistakenly uses the length of the `rewardTokens` state variable to control the loop, which iterates over the `_rewardTokens` input array. This discrepancy causes several issues: - The function will revert if `_rewardTokens.length` is less than `rewardTokens.length`, due to an out-of-bounds access on `_rewardTokens`. - The function does not iterate over the entire `_rewardTokens` list if `_rewardTokens.length` is greater than `rewardTokens.length`. - If `_rewardTokens.length` equals `rewardTokens.length`, the function processes all elements, but this does not address the core issue mentioned in the previous audit.

Impact

The function's incorrect loop control can cause reverts, halting execution and preventing updates to unseen rewards. This flaw undermines the functionality of filtering and processing reward tokens, potentially leading to missed rewards and contract instability.

Code Snippet

```
function notifyUnseenReward(address[] memory _rewardTokens) external {
    uint256 length = rewardTokens.length;
    for (uint256 i = 0; i < length; ++i) {
        if (rewardTokenAdded[_rewardTokens[i]]) {
            _notifyUnseenReward(_rewardTokens[i]);
        }
    }
}
```

Proof Of Concept

For example, if in the contract we have `rewardTokens.length` equal to 10 and we give a list of addresses `_rewardTokens` with the length of 3, the for cycle will have a limit of: `uint256 length = rewardTokens.length;` or equal to 10. On run 4, `i` will be equal to 3 and it will revert as `_rewardTokens[3]` does not exist and is an overflow.

Tool used

Manual Review

Recommendation

Modify the function to use the length of the `_rewardTokens` input array for controlling the loop. Additionally, consider adding a check to ensure `_rewardTokens.length` does not exceed `rewardTokens.length`.

This will look like that:

```
function notifyUnseenReward(address[] memory _rewardTokens) external {
-   uint256 length = rewardTokens.length;
+   uint256 length = _rewardTokens.length;

+   require(length <= rewardTokens.length, "Input exceeds reward tokens
    limit");
    for (uint256 i = 0; i < length; ++i) {
        if (rewardTokenAdded[_rewardTokens[i]]) {
            _notifyUnseenReward(_rewardTokens[i]);
        }
    }
}
```

Low

[L-1] Ownership dependency

Summary

The vulnerability lies in the interaction between the `Lock.sol` and `LockList.sol` contracts. Specifically, if `Lock.sol` is not the owner of the `LockList.sol` contract, certain functions such as `exitLateById` will not function as intended. This is due to the fact that within the `exitLateById` function, there is a call to `locklist.setExitedLateToTrue(msg.sender, id)`, which expects the caller to be the owner of `LockList.sol`. Similar issues may arise in other methods of `Lock.sol` that utilize `onlyOwner` methods of `LockList.sol`.

Vulnerability Detail

The vulnerability arises from the assumption that `Lock.sol` is the owner of `LockList.sol`, leading to improper function calls in cases where this assumption does not hold true.

Impact

The impact of this vulnerability is that certain functionalities within `Lock.sol` may not work as expected if it is not the owner of `LockList.sol`. This could potentially lead to unexpected behaviour or failure of intended operations.

Code Snippet

None

Tool used

Manual Review

Recommendation

Ensure that the owner of `LockList.sol` is the owner of `Lock.sol` also.

[L-2] Unused Library

Summary

The AddressPagination library within the contract remains unused throughout the codebase. It is advisable to remove this library to streamline the contract and reduce unnecessary complexity.

Vulnerability Detail

The AddressPagination library is present in the contract but is not utilized in any of the contract's functions or logic.

Impact

The presence of unused code increases the contract's size and complexity without providing any tangible benefit. Removing the AddressPagination library can declutter the contract codebase, making it easier to maintain and audit.

Tool used

Manual Review

Informational

[I-1] No Zero Checks

Summary

The audit identifies three critical areas in the `Lock` contract where essential checks are missing, potentially leading to unexpected behaviour or vulnerabilities.

Vulnerability Detail

1. **Missing `address(0)` Check in `Lock::setTreasury`:** The `setTreasury` function does not include a check to prevent setting the treasury address to the zero address (`address(0)`), which can lead to loss of funds or inability to interact with the treasury properly.
2. **No Check for Zero Value in `Lock::setDefaultRelockTime`:** The `setDefaultRelockTime` function does not verify whether the `_defaultRelockTime` parameter is zero. Setting the default relock time to zero might lead to logical errors or unintended locking behaviour.
3. **No Check for Zero Values in `Lock::setPenaltyCalcAttributes`:** The `setPenaltyCalcAttribute` function lacks checks to ensure that `_basePenaltyPercentage` and `_timePenaltyFraction` are not zero. Allowing zero values for these parameters could result in incorrect penalty calculations and undermine the penalty mechanism.

Impact

1. **`setTreasury`:** Setting the treasury address to `address(0)` could lead to the inability to manage funds or execute treasury-related functions, which might result in a critical loss of functionality.
2. **`setDefaultRelockTime`:** Allowing a zero value for the default relock time could cause locking mechanisms to malfunction, potentially allowing locks to be bypassed or not enforced correctly.
3. **`setPenaltyCalcAttributes`:** Zero values for penalty calculation parameters could disable the intended penalty logic, leading to incorrect fee assessments and reducing the effectiveness of the penalty system.

Code Snippet

```
function setTreasury(address _treasury) external onlyOwner {
    treasury = _treasury;
    emit SetTreasury(_treasury);
}

function setDefaultRelockTime(uint256 _defaultRelockTime) external
    onlyOwner {
    defaultRelockTime = _defaultRelockTime;
}

function setPenaltyCalcAttributes(uint256 _basePenaltyPercentage,
    uint256 _timePenaltyFraction) external onlyOwner {
    if (_basePenaltyPercentage > WHOLE || _timePenaltyFraction > WHOLE)
        revert WrongScaledPenaltyAmount();
    basePenaltyPercentage = _basePenaltyPercentage;
    timePenaltyFraction = _timePenaltyFraction;
    emit SetPenaltyCalcAttribute(_basePenaltyPercentage,
        _timePenaltyFraction);
}
```

Tool used

Manual Review

Recommendation

1. **`setTreasury`:** Add a check to ensure the treasury address is not set to `address(0)`:

```
function setTreasury(address _treasury) external onlyOwner {
    require(_treasury != address(0), "Invalid treasury address");
    treasury = _treasury;
    emit SetTreasury(_treasury);
}
```

2. **`setDefaultRelockTime`:** Add a check to ensure `_defaultRelockTime` is not zero:

```
function setDefaultRelockTime(uint256 _defaultRelockTime) external
    onlyOwner {
     require(_defaultRelockTime != 0, "Invalid default relock time");
     defaultRelockTime = _defaultRelockTime;
}
```

3. **`setPenaltyCalcAttributes`:** Add checks to ensure `_basePenaltyPercentage` and `_timePenaltyFraction` are not zero:

```
function setPenaltyCalcAttributes(uint256 _basePenaltyPercentage,
    uint256 _timePenaltyFraction) external onlyOwner {
     require(_basePenaltyPercentage != 0, "Invalid base penalty
        percentage");
     require(_timePenaltyFraction != 0, "Invalid time penalty fraction")
        ;
     if (_basePenaltyPercentage > WHOLE || _timePenaltyFraction > WHOLE)
         revert WrongScaledPenaltyAmount();
     basePenaltyPercentage = _basePenaltyPercentage;
     timePenaltyFraction = _timePenaltyFraction;
     emit SetPenaltyCalcAttribute(_basePenaltyPercentage,
        _timePenaltyFraction);
}
```

[I-2] Wrong Natspec Docs

Summary

The `Lock::setLockTypeInfo` function's natspec documentation does not accurately reflect its implementation. The documentation suggests that the function updates the lock periods and reward multipliers arrays, while the implementation actually deletes the existing arrays and replaces them with new values.

Vulnerability Detail

The natspec documentation for the `setLockTypeInfo` function states:

```
/// @notice Configures lock periods and their corresponding reward
    multipliers for staking.
/// @dev This function can only be called by the contract owner and is
    used to set or update the lock periods and reward multipliers arrays
    .
/// @param _lockPeriod An array of lock periods in seconds.
/// @param _rewardMultipliers An array of multipliers corresponding to
    each lock period; these multipliers enhance the rewards for longer
    lock periods.
```

However, the implementation of the function is as follows:

```
function setLockTypeInfo(
    uint256[] calldata _lockPeriod,
    uint256[] calldata _rewardMultipliers
) external onlyOwner {
    if (_lockPeriod.length != _rewardMultipliers.length)
        revert InvalidLockPeriod();
    delete lockPeriod;
    delete rewardMultipliers;
    uint256 length = _lockPeriod.length;
    for (uint256 i; i < length; ) {
        lockPeriod.push(_lockPeriod[i]);
        rewardMultipliers.push(_rewardMultipliers[i]);
        unchecked {
            i++;
        }
    }

    emit SetLockTypeInfo(lockPeriod, rewardMultipliers);
}
```

In the function implementation, the lines:

```
delete lockPeriod;
delete rewardMultipliers;
```

delete the old values of the arrays before setting the new ones. This means that the function does not update the arrays but rather replaces them entirely.

Impact

This discrepancy between the documentation and the implementation can mislead developers or auditors who rely on the documentation for understanding the function's behavior. It may cause incorrect assumptions about the state of the arrays before and after the function call, potentially leading to bugs or unexpected behavior in the system that depends on this function.

Code Snippet

```
function setLockTypeInfo(
    uint256[] calldata _lockPeriod,
    uint256[] calldata _rewardMultipliers
) external onlyOwner {
    if (_lockPeriod.length != _rewardMultipliers.length)
        revert InvalidLockPeriod();
    delete lockPeriod;
    delete rewardMultipliers;
    uint256 length = _lockPeriod.length;
    for (uint256 i; i < length; ) {
        lockPeriod.push(_lockPeriod[i]);
        rewardMultipliers.push(_rewardMultipliers[i]);
```

```
            unchecked {
                i++;
            }
        }

        emit SetLockTypeInfo(lockPeriod, rewardMultipliers);
    }
```

Tool used

Manual Review

Recommendation

Update the natspec documentation to accurately reflect the behavior of the function. The corrected documentation should state that the function sets the lock periods and reward multipliers arrays, replacing any existing values.

Suggested natspec documentation update:

```
/// @notice Sets the lock periods and their corresponding reward
    multipliers for staking.
/// @dev This function can only be called by the contract owner and is
    used to set the lock periods and reward multipliers arrays,
    replacing any existing values.
/// @param _lockPeriod An array of lock periods in seconds.
/// @param _rewardMultipliers An array of multipliers corresponding to
    each lock period; these multipliers enhance the rewards for longer
    lock periods.
```

Gas

Summary

This vulnerability concerns gas optimization within the `Lock::_updateRewardDebt` function. Specifically, the length of the `rewardTokens` array in `Lock::_updateRewardDebt` can be extracted as a variable to improve gas efficiency. Additionally, the increment operation `++i` can be executed within an unchecked block for further optimization. The vulnerability in the `_getReward` function of `Lock.sol` involves multiple executions of the calculation `reward / 1e3`. It is recommended to extract this calculation to execute only once for optimization purposes.

Vulnerability Detail

The gas optimization issue involves the unnecessary computation of `rewardTokens.length` within the `_updateRewardDebt` function. Extracting this value as a variable would streamline gas usage. Furthermore, the increment operation `++i` can be performed within an unchecked block to reduce gas consumption. Within the `_getReward` function of `Lock.sol`, the expression `reward / 1e3` is computed multiple times, leading to redundant calculations and potential gas inefficiencies.

Code Snippet

```
function _updateRewardDebt(address _user) internal {
        Balances memory bal = balances[_user]; // Retrieve the current
            balance information for the user.

        for (uint i = 0; i < rewardTokens.length; ++i) {
            address rewardToken = rewardTokens[i]; // Access each
                reward token.
            Reward memory rewardInfo = rewardData[rewardToken]; // Get
                the current reward data for each token.

            // Recalculate the reward debt for the user based on their
                locked balances and the accumulated rewards for the
                token.
            rewardDebt[_user][rewardToken] = rewardInfo.cumulatedReward
                 * bal.lockedWithMultiplier;
        }
    }
```

Impact

Failure to optimize gas usage can lead to inefficiencies in contract execution, potentially resulting in higher gas costs for transactions involving the affected functions. While this may not pose a critical security risk, it can impact the usability and cost-effectiveness of the contract. The repeated calculation of `reward / 1e3` in `_getReward` can result in increased gas costs for transactions involving this function. Extracting this calculation to execute only once can improve gas efficiency and optimize contract execution.

Tool used

Manual Review