SoulMate Security Audit Report by MK (16th DEC 2024)

High Risk Findings

[H-1] Users don't need to lock tokens in Staking contract to get rewards

Description: A user can use `Staking::claimRewards()` without having to lock tokens in the staking contract.

Impact: Users can claim rewards of staking without having to lock tokens in the contract severely breaking the protocol and opening up to several exploits such as using a flashloan to acquire big amounts of `Love Token` deposit in the staking contract call `claimRewards()`, and withdraw the tokens.

Proof of Concept:

1. After minting NFT users, start accruing rewards without depositing, they can then call `Staking::deposit()` to deposit tokens, and immediately after call `Staking::claimRewards()` to claim rewards corresponding to the period since they minted the NFT, and withdraw tokens immediately after. This process can be repeated for subsequential `claimRewards`, users can deposit immediately before claiming rewards and then withdraw.

```
    function test_user_can_claim_rewards_without_waiting_staking() public {
        uint256 weekOfStaking = 5;
        _mintOneTokenForBothSoulmates();

        vm.warp(block.timestamp + weekOfStaking * 1 weeks + 1 seconds);
        console2.log(loveToken.balanceOf(soulmate1));

        uint256 initialTokenBalance = 10 ether;
        deal(address(loveToken), soulmate1, initialTokenBalance);
        vm.startPrank(soulmate1);
        loveToken.approve(address(stakingContract), type(uint256).max);
        stakingContract.deposit(initialTokenBalance);
        stakingContract.claimRewards();
        stakingContract.withdraw(initialTokenBalance);
        vm.stopPrank();

        assertTrue(loveToken.balanceOf(soulmate1) == weekOfStaking *
initialTokenBalance + initialTokenBalance);

        vm.warp(block.timestamp + weekOfStaking * 1 weeks + 1 seconds);
        vm.startPrank(soulmate1);
        loveToken.approve(address(stakingContract), type(uint256).max);
        stakingContract.deposit(initialTokenBalance);
        stakingContract.claimRewards();
        stakingContract.withdraw(initialTokenBalance);
        vm.stopPrank();
        assertTrue(loveToken.balanceOf(soulmate1) == 2 * weekOfStaking *
initialTokenBalance + initialTokenBalance);
    }
```

Recommended Mitigation:

1. This issue can be mitigated by making it so that when users deposit tokens the `Staking::lastClaim` mapping is also updated.

```
    function deposit(uint256 amount) public {
        if (loveToken.balanceOf(address(stakingVault)) == 0) {
            revert Staking__NoMoreRewards();
        }
        // No require needed because of overflow protection
+       _claimRewards();
+       lastClaim[msg.sender] = block.timestamp;
        userStakes[msg.sender] += amount;
        loveToken.transferFrom(msg.sender, address(this), amount);

        emit Deposited(msg.sender, amount);
    }
```

2. However, this makes it so that any pending rewards for the user are lost, so we can create an internal `_claimRewards()` function to remove any pending rewards.

```
 function _claimRewards() internal {
        uint256 soulmateId = soulmateContract.ownerToId(msg.sender);
        // first claim
        if (lastClaim[msg.sender] == 0){
            return;
        }

        // How many weeks passed since the last claim.
        // Thanks to round-down division, it will be the lower amount possible
until a week has completly pass.
        uint256 timeInWeeksSinceLastClaim = ((block.timestamp -
lastClaim[msg.sender]) / 1 weeks);
        if (timeInWeeksSinceLastClaim < 1) {
            return;
        }

        lastClaim[msg.sender] = block.timestamp;

        // Send the same amount of LoveToken as the week waited times the number
of token staked
        uint256 amountToClaim = userStakes[msg.sender] *
timeInWeeksSinceLastClaim;
        loveToken.transferFrom(address(stakingVault), msg.sender, amountToClaim);
        emit RewardsClaimed(msg.sender, amountToClaim);
    }
```

[H-2] Lack of validations at `Airdrop::claim()` allows attackers to claim lovetokens without a Soulbound NFT

Description:

1. The `Airdrop::claim()` function lacks validations to prevent token claiming for users without a Soulbound NFT. This enables attackers to call the `claim` function and obtain tokens as if they had the first minted Soulbound NFT.

2. Additionally, there is a flaw in the use of the `Soulmate::isDivorced` function, where the airdrop contract address is used instead of the user's address. A divorced user can still claim a Soulbound NFT.

Impact:

1. An attacker lacking a Soulbound NFT can claim rewards, receiving them as if they have the first minted NFT.

2. Here's a test demonstrating two separate claims with a 10-day gap between each.

```
    function testClaimNoSoulmate() public {
        console.log("Balance before: ", loveToken.balanceOf(soulmate1) / 10 **
loveToken.decimals());

        vm.warp(block.timestamp + 10 days + 1 seconds);

        vm.prank(soulmate1);
        airdropContract.claim();

        console.log("Balance after claim 1: ", loveToken.balanceOf(soulmate1) / 10
** loveToken.decimals());

        vm.warp(block.timestamp + 20 days + 1 seconds);

        vm.prank(soulmate1);
        airdropContract.claim();

        console.log("Balance after claim 2: ", loveToken.balanceOf(soulmate1) / 10
** loveToken.decimals());
   }
```

3. Output

```
Logs:
  Balance before:  0
  Balance after claim 1:  10
  Balance after claim 2:  30

```

Proof of Concept:

1. The `isDivorced` check will always pass since it sends the `msg.sender` of the contract to the function, which consistently returns false for the Airdrop contract address.

2. `numberOfDaysInCouple` is computed using the timestamp when the NFT was minted, but for the attacker without a minted NFT, it returns the timestamp of the first minted NFT.

```
    function claim() public {
        // No LoveToken for people who don't love their soulmates anymore.
@>      if (soulmateContract.isDivorced()) revert Airdrop__CoupleIsDivorced();

@>          //No validation here if the msg.sender has a Soulbound NFT

        // Calculating since how long soulmates are reunited
@>      uint256 numberOfDaysInCouple = (block.timestamp -
            soulmateContract.idToCreationTimestamp(
                soulmateContract.ownerToId(msg.sender)
            )) / daysInSecond;


        ...
        ...
    }
```

Recommended Mitigation:

1. Revise the `Airdrop::claim` function to include a check verifying that the caller possesses a minted Soulbound NFT, and start counting from nextID = 1, so an id of 0 will be used as invalid.
2. Adjust the `Soulmate::isDivorced` function to accurately validate the msg.sender.

```
/// @notice Claim tokens. Every person who have a Soulmate NFT token can claim 1
LoveToken per day.
    function claim() public {
        // No LoveToken for people who don't love their soulmates anymore.
-       if (soulmateContract.isDivorced()) revert Airdrop__CoupleIsDivorced();
+       if (soulmateContract.isDivorced(msg.sender)) revert
Airdrop__CoupleIsDivorced();

++      require(soulmateContract.ownerToId(msg.sender) != 0);

        // Calculating since how long soulmates are reunited
        uint256 numberOfDaysInCouple = (block.timestamp -
            soulmateContract.idToCreationTimestamp(
                soulmateContract.ownerToId(msg.sender)
            )) / daysInSecond;
```

```
                ...
                ...
    }
```

3. Update the `isDivorced` function to correctly handle the msg.sender as a parameter.

```
--  function isDivorced() public view returns (bool) {
--      return divorced[msg.sender];
--  }

++  function isDivorced(address soulmate) public view returns (bool) {
++      return divorced[soulmate];
++  }
```

[H-3] Soulmate waiting to reunite can claim an abnormal amount of LoveTokens as staked rewards in StakingVault

Description: Soulmate waiting to reunite can get an unexpectedly high amount of LoveTokens as staking rewards. As the idToCreationTimestamp of the token has not been set yet, it equals zero, the calculation of the period for the stake is then too high, and the reward is miscalculated.

Impact: Users can be rewarded a overevaluated amount of tokens.

Proof of Concept:

1. The attacker calls mintSoulmateToken() after the emission of SoulmateAreReunited (so that he is the first in the couple)
   the attacker then deposit a certain quantity (X) of LoveToken in the staking contract (that he got from any way) The attacker then can claim staking rewards:

```
            lastClaim[msg.sender] = soulmateContract.idToCreationTimestamp(
                soulmateId
            );
```

2. As the `soulmateContract.idToCreationTimestamp(soulmateId)` is not set up yet, this will be equal to 0
   `timeInWeeksSinceLastClaim` is then equal to "(block.timestamp - 0) / 1 weeks" (approx equals to 2823)

3. Then the `uint256 amountToClaim = userStakes[msg.sender] * timeInWeeksSinceLastClaim;` is very high, resulting in

```
loveToken.transferFrom(
            address(stakingVault),
            msg.sender,
            amountToClaim
        );
```

to be a lot higher than expected (X * 2823)

Recommended Mitigation: Check if msg.sender has a soulmate

[H-4] Inaccurate Divorce Status Check in Airdrop Claim Function

Description: The claim function in the Airdrop contract presents a vulnerability wherein divorced individuals can erroneously claim rewards. This vulnerability stems from the function's failure to accurately verify the divorce status of the associated soulmate. Consequently, divorced users can exploit this flaw to claim rewards intended for non-divorced soulmates.

Impact: The vulnerability poses a significant risk to the fairness and integrity of the reward distribution system. By exploiting this flaw, divorced users can claim rewards illegitimately, leading to unfair allocation and potential financial losses.

Proof of Concept:

1. `if (soulmateContract.isDivorced()) revert Airdrop__CoupleIsDivorced();` This line checks if the caller's address (msg.sender) is divorced according to the Soulmate contract. However, the isDivorced() function in the Soulmate contract does not take any arguments, and it always checks the divorce status for msg.sender.

2. Since msg.sender in the Airdrop contract is the address that is calling the claim function, not necessarily the soulmate address associated with the claimed NFT, this check does not accurately determine if the soulmate associated with the claimed NFT is divorced.

```
    function claim() public {
        // No LoveToken for people who don't love their soulmates anymore.
        if (soulmateContract.isDivorced()) revert Airdrop__CoupleIsDivorced();


        // Calculating since how long soulmates are reunited
        uint256 numberOfDaysInCouple = (block.timestamp -
            soulmateContract.idToCreationTimestamp(
                soulmateContract.ownerToId(msg.sender)
            )) / daysInSecond;


        uint256 amountAlreadyClaimed = _claimedBy[msg.sender];


        if (
            amountAlreadyClaimed >=
            numberOfDaysInCouple * 10 ** loveToken.decimals()
        ) revert Airdrop__PreviousTokenAlreadyClaimed();
        uint256 tokenAmountToDistribute = (numberOfDaysInCouple *
            10 ** loveToken.decimals()) - amountAlreadyClaimed;


        // Dust collector
        if (
            tokenAmountToDistribute >=
            loveToken.balanceOf(address(airdropVault))
        ) {
            tokenAmountToDistribute = loveToken.balanceOf(
                address(airdropVault)
            );
        }
        _claimedBy[msg.sender] += tokenAmountToDistribute;


        emit TokenClaimed(msg.sender, tokenAmountToDistribute);


        loveToken.transferFrom(
            address(airdropVault),
            msg.sender,
            tokenAmountToDistribute
        );
    }
```

```
    function testClaimDivorced() public {
        address alice = address(0x1);
        address bob = address(0x2);

        // Alice mints a soulmate token.
        vm.startPrank(alice);
        uint256 aliceTokenId = soulmateContract.mintSoulmateToken();
        vm.stopPrank();

        // Bob mints a soulmate token.
        vm.startPrank(bob);
        uint256 bobTokenId = soulmateContract.mintSoulmateToken();
        vm.stopPrank();

        // Assert that Alice and Bob are now soulmates.
        assertEq(soulmateContract.soulmateOf(alice), bob);
        assertEq(soulmateContract.soulmateOf(bob), alice);

        // Alice initiates a divorce.
        vm.startPrank(alice);
        vm.warp(block.timestamp + 200 days + 1 seconds);
        soulmateContract.getDivorced();

        // Check Alice's divorce status.
        bool isDivorced = soulmateContract.isDivorced();
        assertTrue(isDivorced, "Alice should be divorced.");

        // Expect the claim to fail due to divorce.
        bytes4 expectedRevertReason =
bytes4(keccak256("Airdrop__CoupleIsDivorced()"));
        vm.expectRevert(expectedRevertReason);

        // Attempt to claim should fail.
        airdropContract.claim();

        // Check that Alice did not receive any tokens.
        uint256 aliceBalance = loveToken.balanceOf(alice);
        assertEq(aliceBalance, 0, "Alice should not have received tokens after
divorce");

        vm.stopPrank();
    }
```

Recommended Mitigation:

1. Modify the claim function in the Airdrop contract to accurately check the divorce status of the soulmate associated with the claimed NFT. Here possible fix for this: add or modify the `isDivorced()`as:

```diff
-     function isDivorced() public view returns (bool) {
+    function isDivorced(address soulmate) public view returns (bool) {
-         return divorced[msg.sender];
+         return divorced[soulmate];
    }
```

```diff
    function claim() public {
        // No LoveToken for people who don't love their soulmates anymore.
-         if (soulmateContract.isDivorced()) revert Airdrop__CoupleIsDivorced();
+        if (soulmateContract.isDivorced(msg.sender)) revert
Airdrop__CoupleIsDivorced();

        // Calculating since how long soulmates are reunited
        uint256 numberOfDaysInCouple = (block.timestamp -
            soulmateContract.idToCreationTimestamp(
                soulmateContract.ownerToId(msg.sender)
            )) / daysInSecond;


        uint256 amountAlreadyClaimed = _claimedBy[msg.sender];


        if (
            amountAlreadyClaimed >=
            numberOfDaysInCouple * 10 ** loveToken.decimals()
        ) revert Airdrop__PreviousTokenAlreadyClaimed();


        uint256 tokenAmountToDistribute = (numberOfDaysInCouple *
            10 ** loveToken.decimals()) - amountAlreadyClaimed;


        // Dust collector
        if (
            tokenAmountToDistribute >=
            loveToken.balanceOf(address(airdropVault))
        ) {
            tokenAmountToDistribute = loveToken.balanceOf(
                address(airdropVault)
            );
        }
        _claimedBy[msg.sender] += tokenAmountToDistribute;


        emit TokenClaimed(msg.sender, tokenAmountToDistribute);


        loveToken.transferFrom(
            address(airdropVault),
            msg.sender,
            tokenAmountToDistribute
        );
    }
```

Medium Risk Findings

[M-1] Anyone can write to Soulmate shared space

Description: Anyone can write in the shared space by calling `Soulmate::writeMessageInSharedSpace`.

Impact: Any user will be able to write a message in the space of the owners of token id 0.

Proof of Concept:

1. To write in the shared space the function `writeMessageInSharedSpace`checks the id of the soul token owned by the caller by `uint256 id = ownerToId[msg.sender];`, for a user that doesn't have a soulmate token, this will result in `id = 0` and therefore will be allowed to write a message in the space of the owners of token id 0.

2. Copy the following test into `SoulmateTest.t.sol` and run forge test --mt test_write_in_shares_space_0

```
function test_write_in_shares_space_0() public {
        address random_writter = makeAddr("random_writter");
        vm.prank(random_writter);
        soulmateContract.writeMessageInSharedSpace("Buy some eggs!");

        // anyone can view messages in blockchain
        string memory message = soulmateContract.readMessageInSharedSpace();
        string[4] memory possibleText = [
            "Buy some eggs!, sweetheart",
            "Buy some eggs!, darling",
            "Buy some eggs!, my dear",
            "Buy some eggs!, honey"
        ];
        bool found;
        for (uint256 i; i < possibleText.length; i++) {
            if (compare(possibleText[i], message)) {
                found = true;
                break;
            }
        }
        assertTrue(found);
    }
```

Recommended Mitigation:

1. Change starting tokenId in `Soulmate.sol` to 1 and only allow users with token ID greater than 1 to write messages.

```
+    error Soulmate__dontHaveASoulmate();
...
+    uint256 private nextID = 1;
....
    function writeMessageInSharedSpace(string calldata message) external {
        uint256 id = ownerToId[msg.sender];
+       if(id == 0) revert();
        sharedSpace[id] = message;
        emit MessageWrittenInSharedSpace(id, message);
    }
...
    function totalSupply() external view returns (uint256) {
-        return nextID;
+        return nextID-1;
    }

    function totalSouls() external view returns (uint256) {
-        return nextID * 2;
+        return (nextID-1) * 2;
    }
```

[M-2] User can couple with themself, being able to claim love token. soulmate::mintSoulmateToken

Description: A user can couple with themselves. This can be on purpose or by accident. This allows the user to mint NFT and claim love tokens. This means they can also stake tokens and receive more tokens.

Impact:

1. This will enable tokens to be sent to Users who are not entitled to them. It means that they will not need to couple with someone else. This defeats the purpose of the protocol.

Proof of Concept:

```
**Proof of Concept**
function test_Himself1() public {
        address USER = makeAddr("user");
        vm.startPrank(USER);               // broadcasting  User
        soulmateContract.mintSoulmateToken(); // press mint twice
        soulmateContract.mintSoulmateToken();
        address workCouple = soulmateContract.soulmateOf(USER); //shows soulmate
is same user as couple
        console2.log("the partner of USER1 is:", workCouple);
        vm.stopPrank();
        vm.warp(86401);
        vm.prank(USER);
        airdropContract.claim();

        }
```

Output:

```
    624] Soulmate::soulmateOf(user: [0x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
[staticcall]
    │   └─ ← user: [0x6CA6d1e2D5347Bfab1d91e883F1915560e09129D] //shows partner is
same user


    │   ├─ [2586] LoveToken::balanceOf(Vault:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b])
    │   │   └─ ← 500000000000000000000000000 [5e26]
    │   ├─ emit TokenClaimed(user: user:
[0x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], amount: 1000000000000000000 [1e18])
    │   ├─ [33184] LoveToken::transferFrom(Vault:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], user:
[0x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], 1000000000000000000 [1e18])
    │   │   ├─ emit Transfer(from: Vault:
[0x8Ad159a275AEE56fb2334DBb69036E9c7baCEe9b], to: user:
[0x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], amount: 1000000000000000000 [1e18])
    │   │   └─ ← true
    │   └─ ← ()                             //!!RECEIVING FUNDS!!
    └─ ← ()
```

Recommended Mitigation:

1. Add this line to soulmate::mintSoulmateToken line 75:

```
else if (soulmate2 == address(0)) {
+           require(idToOwners[nextID][0] != msg.sender, "you cant couple with
yourself!");
            idToOwners[nextID][1] = msg.sender;
            // Once 2 soulmates are reunited, the token is minted
            ownerToId[msg.sender] = nextID;
            soulmateOf[msg.sender] = soulmate1;
            soulmateOf[soulmate1] = msg.sender;
            idToCreationTimestamp[nextID] = block.timestamp;
```

result:

```
[33015] Soulmate::mintSoulmateToken()
    │   ├─ emit SoulmateIsWaiting(soulmate: user:
[0x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
    │   └─ ← 0
    ├─ [1375] Soulmate::mintSoulmateToken()
    │   └─ ← revert: you cant couple with yourself!
    └─ ← revert: you cant couple with yourself!
```

Low Risk Findings

[L-1] The event SoulmateAreReunited is triggered with incorrect parameters

Description: The event SoulmateAreReunited is triggered with incorrect parameters.

Impact: Wrong event emission will lead to wrogn end user information and stats.

Proof of Concept: The event SoulmateAreReunited is triggered with incorrect parameters.

```
    function test_MintNewToken() public {
        uint tokenIdMinted = 0;

        vm.prank(soulmate1);
        soulmateContract.mintSoulmateToken();

        assertTrue(soulmateContract.totalSupply() == 0);

        vm.prank(soulmate2);
        soulmateContract.mintSoulmateToken();
        assertTrue(soulmateContract.totalSupply() == 1);
        assertTrue(soulmateContract.soulmateOf(soulmate1) == soulmate2);
        assertTrue(soulmateContract.soulmateOf(soulmate2) == soulmate1);
        assertTrue(soulmateContract.ownerToId(soulmate1) == tokenIdMinted);
        assertTrue(soulmateContract.ownerToId(soulmate2) == tokenIdMinted);
    }
```

result

```
[PASS] test_MintNewToken() (gas: 210743)
Traces:
  [210743] SoulmateTest::test_MintNewToken()
    ├─ [0] VM::prank(soulmate1: [0x65629adcc2F9C857Aeb285100Cc00Fb41E78DC2f])
    │   └─ ← ()
    ├─ [33015] Soulmate::mintSoulmateToken()
    │   ├─ emit SoulmateIsWaiting(soulmate: soulmate1:
[0x65629adcc2F9C857Aeb285100Cc00Fb41E78DC2f])
    │   └─ ← 0
    ├─ [393] Soulmate::totalSupply() [staticcall]
    │   └─ ← 0
    ├─ [0] VM::prank(soulmate2: [0xe93A5E9F20AF38E00a08b9109D20dEc1b965E891])
    │   └─ ← ()
    ├─ [157343] Soulmate::mintSoulmateToken()
    │   ├─ emit SoulmateAreReunited(soulmate1: soulmate1:
[0x65629adcc2F9C857Aeb285100Cc00Fb41E78DC2f], soulmate2:
0x0000000000000000000000000000000000000000, tokenId: 0)
    │   ├─ emit Transfer(from: 0x0000000000000000000000000000000000000000, to:
soulmate2: [0xe93A5E9F20AF38E00a08b9109D20dEc1b965E891], id: 0)
```

Recommended Mitigation:

```
    emit SoulmateAreReunited(soulmate1, msg.sender, nextID);
```

[L-2] Staking Rewards Forfeited Upon Deposit Withdrawal

Description: `Staking.sol` is designed to reward users for locking their love tokens over a period. However, due to a flaw in the implementation, users who withdraw their deposited funds before claiming their accrued rewards forfeit these rewards. This behavior deviates from the expected outcome where rewards accrued during the staking period should be claimable until the moment of withdrawal.

Impact:

1. Loss of Expected Rewards: Users lose potential rewards they have rightfully earned through staking, which can lead to dissatisfaction and reduced participation in the staking mechanism.
2. Misalignment with Staking Incentives: The fundamental incentive for staking — earning rewards over time — is undermined if users risk losing rewards by withdrawing their stake.

Proof of Concept: The vulnerability stems from the Staking contract's handling of reward claims and withdrawals. Specifically, the contract does not account for or preserve the rewards accrued by a user's stake when they perform a withdrawal. As demonstrated in the provided test code, after advancing time to allow for reward accrual and then withdrawing the initial deposit, the user's balance before and after attempting to claim rewards remains unchanged, indicating that the rewards were forfeited upon withdrawal.

```
    function test_rewardsForfeitedUponDepositWithdrawal() public {
        uint256 balanceUser1;
        uint256 balanceUser1_beforeRewardsClaim;
        uint256 balanceUser1_afterRewardsClaim;
        uint256 totalAmountDeposited = 0;

        // pair up user 1 and 2
        vm.prank(user1);
        soulmateContract.mintSoulmateToken();
        vm.prank(user2);
        soulmateContract.mintSoulmateToken();

        vm.warp(block.timestamp + 4 weeks); // fast fwd time with 1 month
        vm.startPrank(user1);
        airdropContract.claim(); // claim airdrop
        balanceUser1 = loveToken.balanceOf(user1);
        totalAmountDeposited += balanceUser1;
        loveToken.approve(address(stakingContract), type(uint256).max); // approve
spending
        stakingContract.deposit(balanceUser1); // deposit all
        vm.stopPrank();

        vm.warp(block.timestamp + 4 weeks); // fast fwd time with 1 month (due to
another bug, not really neccesary)

        vm.startPrank(user1);
        stakingContract.withdraw(totalAmountDeposited);
        balanceUser1_beforeRewardsClaim = loveToken.balanceOf(user1); // no
rewards to claim
        stakingContract.claimRewards();
        balanceUser1_afterRewardsClaim = loveToken.balanceOf(user1);
        vm.stopPrank();

        assertEq(balanceUser1_beforeRewardsClaim, balanceUser1_afterRewardsClaim);
    }
```

Recommended Mitigation:

1. Automate the rewards claim process during withdrawal to eliminate the need for users to manually claim rewards in a separate transaction:

```
    function withdraw(uint256 amount) public {
+     claimRewards();
        // No require needed because of overflow protection
        userStakes[msg.sender] -= amount;
        loveToken.transfer(msg.sender, amount);
        emit Withdrew(msg.sender, amount);
    }
```

2. Additionally, do clearly document and communicate this change to users, explaining how the automatic reward claim process works and its benefits, to maintain transparency and trust.

[L-3] Misleading Total Souls Count due to Unpaired and Self-Paired Users

Description: `Soulmate::totalSouls` inaccurately calculates the total number of paired souls. This function currently returns a value that assumes all minting actions result in successful pairings, thereby doubling the `nextID` to represent total souls. However, this calculation does not account for users who are still awaiting pairing or those who have been incorrectly paired with themselves, leading to a misrepresented count of active soulmate pairs within the system.

Impact: The inaccurate reporting of total souls impacts the transparency and reliability of the protocol's metrics. It could mislead users and stakeholders about the platform's activity level and the actual number of successful pairings, potentially affecting user trust and engagement.

Proof of Concept: The `totalSouls` function's simplistic calculation method (return nextID * 2;) overlooks two critical scenarios:

1. Unpaired Souls: Users who have initiated the minting process but are not yet paired. These users should not contribute to the total count of souls until their pairing is confirmed.
2. Self-Paired Users: The current logic does not prevent a user from being paired with themselves.

The provided test case illustrates this issue by demonstrating that the `totalSouls` count can be inaccurate immediately after a minting request and can also reflect an incorrect increment when a user is allowed to pair with themselves.

```
    function test_numberOfSouls() public {
        uint256 totalSouls = soulmateContract.totalSouls();
        console2.log(totalSouls);
        assertEq(totalSouls, 0);

        vm.prank(user1);
        soulmateContract.mintSoulmateToken();
        totalSouls = soulmateContract.totalSouls();
        console2.log(totalSouls);
        assertLt(totalSouls, 1);

        vm.prank(user1);
        soulmateContract.mintSoulmateToken();
        totalSouls = soulmateContract.totalSouls();
        console2.log(totalSouls);
        assertGt(totalSouls, 1);
    }
```

Recommended Mitigation:

1. Prevent Self-Pairing: Implement checks within the minting function to prevent users from being paired with themselves, ensuring that all pairings are between distinct users:

```
  function mintSoulmateToken() public returns (uint256) {
        // Check if people already have a soulmate, which means already have a
token
        address soulmate = soulmateOf[msg.sender];
        if (soulmate != address(0)) {
            revert Soulmate__alreadyHaveASoulmate(soulmate);
        }
        address soulmate1 = idToOwners[nextID][0];
        address soulmate2 = idToOwners[nextID][1];
        if (soulmate1 == address(0)) {
            idToOwners[nextID][0] = msg.sender;
            ownerToId[msg.sender] = nextID;
            emit SoulmateIsWaiting(msg.sender);
        } else if (soulmate2 == address(0)) {
+          require(msg.sender != soulmate1, "Can't be your own soulmate!");
            idToOwners[nextID][1] = msg.sender;
            // Once 2 soulmates are reunited, the token is minted
            ownerToId[msg.sender] = nextID;
            soulmateOf[msg.sender] = soulmate1;
            soulmateOf[soulmate1] = msg.sender;
            idToCreationTimestamp[nextID] = block.timestamp;

            emit SoulmateAreReunited(soulmate1, soulmate2, nextID);

            _mint(msg.sender, nextID++);
        }

        return ownerToId[msg.sender];
    }
```

2. Rename the function and change its implementation so that it returns the number of pairs, not the number of souls:

```
-    function totalSouls() external view returns (uint256) {
-        return nextID * 2;
+    function totalPairs() external view returns (uint256) {
+        return nextID;
    }
```

[L-4] `Soulmate::_mint()` can cause NFT to be frozen in a contract if not supported by soulmate's contract

Description: `Soulmate::_mint()` can cause NFT to be frozen in a contract if not supported by soulmate's contract.

Impact:

Proof of Concept:

1. In `Soulmate::mintSoulmateToken()` function it calls the `_mint` function from `ERC721` contract by openZeppelin to mint a Soulbound Token for the newly formed soulmates.
2. However if the soulmate's contract does not support ERC721 tokens it could result in the NFT being frozen in the contract.
3. As per the documentation of EIP-721:

   A wallet/broker/auction application MUST implement the wallet interface if it will accept safe transfers.

   Ref: https://eips.ethereum.org/EIPS/eip-721

As per the documentation of ERC721.sol by Openzeppelin

Recommended Mitigation: It is adviced to use `ERC721::_safeMint()` instead of `ERC721::_mint()`.

[L-5] Wrong Data emitted in events of `LoveToken`

Description: The `initVault` function in the `LoveToken` contract emits both the `AirdropInitialized` and `StakingInitialized` events with incorrect data. The event data indicates that the parameters passed should be `airdropContract` and `stakingContract`, respectively. Still, the function emits both events with the parameter `managerContract`, resulting in inaccurate event logs.

Impact: The incorrect event data poses a challenge for developers and external systems relying on event logs, as the information provided does not accurately reflect the initialized contracts. This discrepancy may result in confusion and hinder proper tracking of contract initialization events.

Proof of Concept: Both the `AirdropInitialized` and `StakingInitialized` events are expected to log the addresses of the corresponding contracts (`airdropContract` and `stakingContract`). However, the `initVault` function incorrectly emits both events with the parameter `managerContract`, leading to a discrepancy between the event's data and the actual initialized contracts.

Recommended Mitigation:

- Correct Event Data: Update the `initVault` function to emit both the `AirdropInitialized` and `StakingInitialized` events with the correct parameters, reflecting the addresses of the corresponding contracts (`airdropContract` and `stakingContract`).