# Soulmate Audit Report

Version 1.0

*Keyword*

February 11, 2024

# Soulmate Audit Report

xKeywordx

February 11, 2024

Prepared by: xKeywordx

## Table of Contents



## Protocol Summary

Valentine’s day is approaching, and with that, it’s time to meet your soulmate!

We’ve created the Soulmate protocol, where you can mint your shared Soulbound NFT with an unknown person, and get `LoveToken` as a reward for staying with your soulmate. A staking contract is available to collect more love. Because if you give love, you receive more love.

## Disclaimer

Keyword 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 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.

### Scope

```
-- interfaces
|   -- ILoveToken.sol
|   -- ISoulmate.sol
|   -- IStaking.sol
|   -- Ivault.sol
-- protocol
|   -- Airdrop.sol
|   -- LoveToken.sol
|   -- Soulmate.sol
|   -- Staking.sol
|   -- Vault.sol
```

### Roles

None

# Executive Summary

## Known issues

- Eventually, the counter used to give ids will reach the type(uint256).max and no more will be able to be minted. This is known and can be ignored.

## Issues found

| Severity | Number of issues found |
|---|---|
| High | 4 |
| Medium | 0 |
| Low | 0 |
| Gas | 0 |
| Info | 0 |
| Total | 4 |

# Findings

## Highs

### [H-1] Staking rewards exceed `Staking.sol` contract balance leading to stuck user funds

**Description:** In the current implementation of the `Staking.sol` contract, there is an issue where user funds could become stuck due to the contract attempting to distribute more staking rewards than the balance it holds. The contract calculates rewards based on the number of tokens staked and the time they have been staked, without adequately checking whether sufficient funds are available to cover these rewards.

**Impact:** If the contract's balance is insufficient to cover the calculated rewards, any call to the `claimRewards` function will fail, preventing the claiming of rewards.

**Proof of concept:** Add this function to the existing `StakingTest.t.sol` file.

```
1     function testRewardsExceedContractBalance() public {
2         // Step 1: Deposit Tokens
```

```
        uint256 amountToStake = 100000 ether; // Large amount to stake
        uint256 weeksToFastForward = 50; // Number of weeks to simulate

        _depositTokenToStake(amountToStake);

        // Step 2: Fast Forward Time
        vm.warp(block.timestamp + weeksToFastForward * 1 weeks);

        // Check the expected rewards and ensure they exceed the
           contract's balance
        uint256 expectedRewards = amountToStake * weeksToFastForward;
        console2.log("ExpectedRewards: ", expectedRewards);
        uint256 contractBalance = loveToken.balanceOf(address(
           stakingContract));
        console2.log("Contractbalance: ", contractBalance);
        require(contractBalance < expectedRewards, "Contract has enough
            funds, adjust the test scenario");

        // Step 3: Attempt to Claim Rewards, expecting revert
        vm.startPrank(soulmate1);
        vm.expectRevert();
        stakingContract.claimRewards();
        vm.stopPrank();
    }
```

**Recommended mitigation:** To prevent this issue, the Staking contract should implement a check before attempting to distribute rewards to ensure that the contract's balance is sufficient to cover the rewards. If the balance is not sufficient, the contract could proportionally reduce the rewards to match the available balance or prevent the reward claim until the balance is replenished.

### [H-2] `Staking.sol::deposit` function will allow users to deposit even if the contract doesn't have enough balance to pay out rewards.

**Description:** The contract doesn't check to make sure that it has enough balance to pay out the rewards before allowing users to deposit their tokens. This ties somehow to the first reported bug [H-1].

**Impact:** The contract allows a user to deposit tokens as long as the balance of `stakingVault` is not 0. This means that if the vault has 1000 tokens left, and a user deposits 1001, then this user will not be able to use the `claimRewards` function, as this will revert, making the staking feature worthless.

**Proof of concept:** Add this function to the existing `StakingTest.t.sol` file.

```
    function deposit(uint256 amount) public {
-       if (loveToken.balanceOf(address(stakingVault)) == 0) revert
    Staking__NoMoreRewards();
```

```
+       if (loveToken.balanceOf(address(stakingVault)) < amount) revert
    Staking__AmountExceedsRewards();
        // No require needed because of overflow protection
        userStakes[msg.sender] += amount;
        loveToken.transferFrom(msg.sender, address(this), amount);

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

**Recommended mitigation:** To prevent this issue, the `Staking.sol::deposit` function should check to make sure that at the time of depositing, it has enough rewards to repay the amount. Also, I recommend implementing a mechanism for auto-replenishing once in a while so that the balance of the staking contract never runs out of funds to send out rewards.

### [H-3] `Soulmate.sol::getDivorced` function does not update `soulmateOf`, `idToOwners`, or `ownerToId` mappings.

**Description:** The `Soulmate.sol::getDivorced` function marks both parties as divorced, but it does not update `soulmateOf`, `idToOwners`, or `ownerToId` mappings to reflect this change in state. Therefore, even after divorce, the contract still considers the parties as connected soulmates in terms of token ownership and permissions

**Impact:** Divorced soulmates will still accrue staking rewards and are able to use the `Soulmate.sol::writeMessageInSharedSpace` and `Soulmate.sol::readMessageInSharedSpace` functions.

**Proof of concept:** Add this test into the `SoulmateTest.t.sol` file. In order for the test to work, you will need to also change the `BaseTest.t.sol::_giveLoveTokenToSoulmates` function.

```
    function _giveLoveTokenToSoulmates(uint amount) internal {
-       _mintOneTokenForBothSoulmates();
        uint numberDays = amount / 1e18;
        vm.warp(block.timestamp + (numberDays * 1 days));

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

        vm.prank(soulmate2);
        airdropContract.claim();
    }
```

We are doing this because inside the test we mint the NFTs manually before divorcing. The `_depositTokenToStake` function calls this `_giveLoveTokenToSoulmates` function, which calls `_mintOneTokenForBothSoulmates` again, and the test will revert with the `Soulmate.sol::Soulmate__alreadyHaveASoulmate` error. Essentially it tries to mint NFTs again, and

the test will not go through. By temporarily removing this line we can avoid this error, and you can see that divorced soulmates will still accrue staking rewards.

```
    function testDivorcedSoulmatesGetStakingRewards() public {
        uint256 balancePerSoulmates = 5 ether;
        uint weekOfStaking = 5;

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

        vm.prank(soulmate2);
        soulmateContract.mintSoulmateToken();

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

        vm.prank(soulmate2);
        soulmateContract.getDivorced();

        _depositTokenToStake(balancePerSoulmates);

        vm.warp(block.timestamp + weekOfStaking * 1 weeks + 1 seconds);

        vm.prank(soulmate1);
        stakingContract.claimRewards();

        assertTrue(loveToken.balanceOf(soulmate1) == weekOfStaking *
            balancePerSoulmates);

        vm.prank(soulmate1);
        stakingContract.withdraw(balancePerSoulmates);
    }
```

**Recommended mitigation:** Ensure that the `Soulmate.sol::getDivorced` function marks both parties as divorced, and updates `soulmateOf`, `idToOwners`, or `ownerToId` mappings to reflect this change in state.

### [H-4] `Vault.sol` can be drained with a flash loan attack on the `Staking.sol` staking pool.

**Description:** A malicious user could leverage a flash loan in order to drain all the funds from the vault by exploiting the logic in the staking pool. The `Staking.sol::claimRewards` function checks the amount that a user is eligible to withdraw based on the amount deposited multiplied by the number of weeks the user has staked.

**Impact:** All the funds from the vault could be drained.

**Proof of concept:**

1. Imagine a pool that has 1999 tokens.
2. Attacker deposits 1 token.
3. Two weeks pass.
4. Attacker takes a flash loan and buys 2000 tokens.
5. Attacker deposits the funds.
6. Now the pool holds 4000 tokens, and the attacker is eligible to claim 2001 (total tokens deposited) times 2 weeks, because he staked his first token 2 weeks ago, which means 4002 tokens.
7. Attacker calls `claimRewards` function, and drains the vault before returning the flash loan.
8. Pool has 0 tokens now, and the attacker has 2000 tokens after returning the flash loan.

**Recommended mitigation:** A potential mitigation strategy involves tracking the amount staked and the time it was staked for, ensuring rewards are calculated based on “staked time” rather than only looking at the current deposited amount. This involves rethinking the logic of the function