LeaderCat Security Review

Version 2.0

26.12.2024

Conducted by:

NHristov, Junior Security Researcher

Table of Contents

1 About NHristov

[NHristov) is an intern security researcher.

2 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Overview

Project Name	LeaderCat
Repository	https://github.com/LeaderCat/Leader-Cat/tree/main
Commit hash	e312f7a36bcb16b3dc7bb585356e8b423986fa76
Resolution	69e9825cdf8d474f7f1c9ff1f7622da76b62a2c6
Documentation	N/A
Methods	Manual review

Scope

Leader-Cat/LEADERCATCONTRACT.sol

Issues Found

Critical risk	0
High risk	1
Medium risk	0
Low risk	1
Informational	6

5 Findings

5.1 High

5.1.1 [H-1] Incorrect Fee Calculation in LEADERCATCONTRACT:transfer Leads to bigger fees

Description

The `LEADERCATCONTRACT:transfer` function incorrectly calculates the fees for both buy and sell transactions. The fee calculation formula used is:

```
fee = (_amount * fee) / MAX_FEE_PERCENTAGE;
```

This formula does not correctly apply the intended percentage fee, leading to unexpected token distribution. Specifically, the fee calculation should be based on a percentage of the transaction amount, but the current formula results in incorrect fee values.

Impact

The incorrect fee calculation impacts both buyers and sellers, leading to discrepancies in the expected token amounts received after transactions. This can cause users to receive fewer tokens than expected.

Proof of Concepts The steps to reproduce the problem when buying are as follows: 1. Set new fees via `LEADERCATCONTRACT::setFees`e.g. (10% for buys and 20% for sells) 2. Update the dex status of an address with already existing tokens via the function `LEADERCATCONTRACT::updateDEXStatus` 3. From the dex address transfer tokens to another address via the function `LEADERCATCONTRACT::transfer` (e.g. with 100 tokens) 4. Check the existing amount on the `_to` address from the previous transfer is less than expect (80 rather than 90)

PoC Place this in a new file under the test folder in a foundry project

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {LEADERCATCONTRACT} from "../src/LeaderContract.sol";

contract LEADERCATCONTRACTTest is Test {
    LEADERCATCONTRACT public leaderCatContract;

    address owner;
    address feeRecipient;

    function setUp() public {
        feeRecipient = makeAddr("0x123");

        vm.prank(owner);
        uint256 timelockDurationDefault = 24 hours;
        leaderCatContract = new LEADERCATCONTRACT(timelockDurationDefault,
           feeRecipient);
    }

    function testIncorrectFeesAmountForSeller() public {
        uint256 buyFee = 10; // should be 10 percent of the transaction amount
        uint256 sellFee = 20; // should be 20 percent of the transaction amount

        // skip the required time for initial set of the fees
        skip(24 hours);

        address alice = makeAddr("0x1234567890123456789012345678901234567890");
        address dex1 = makeAddr("0x1234567890123456789012345678901234567891");

        vm.startPrank(owner);
        leaderCatContract.setFees(buyFee, sellFee);
        leaderCatContract.transfer(dex1, 1000);

        leaderCatContract.updateDEXStatus(dex1, true);
        vm.stopPrank();

        // fees should be set correctly
        assertEq(leaderCatContract.buyFee(), buyFee);
        assertEq(leaderCatContract.sellFee(), sellFee);

        // alice buys 100 tokens from dex1
        vm.prank(dex1);
        leaderCatContract.transfer(alice, 100);

        // alice should receive 90 tokens but instead she receives 80
        // becase of the formula applied
        // fee = (amount * fee) / MAX_FEE_PERCENTAGE
        // fee = (100 * 10) / 50 = 20
        // alice receives 100 - 20 = 80
        assertEq(leaderCatContract.balanceOf(alice), 80);
    }

    function testIncorrectFeesAmountForBuyer() public {
        uint256 buyFee = 10; // should be 10 percent of the transaction amount
        uint256 sellFee = 20; // should be 20 percent of the transaction amount

        // skip the required time for initial set of the fees
        skip(24 hours);

        address dex1 = makeAddr("0x1234567890123456789012345678901234567811");

        vm.startPrank(owner);

        leaderCatContract.setFees(buyFee, sellFee);

        // check fees applied
        assertEq(leaderCatContract.buyFee(), buyFee);
        assertEq(leaderCatContract.sellFee(), sellFee);

        leaderCatContract.updateDEXStatus(dex1, true);

        // after setting the dex address and status we transfer 1000 tokens to the
           dex
        // this should apply the sell fee logic
        leaderCatContract.transfer(dex1, 1000);
        // dex1 should receive 800 tokens but instead it receives 600
        // becase of the formula applied
        // fee = (amount * fee) / MAX_FEE_PERCENTAGE
        // fee = (1000 * 20) / 50 = 400
        // dex1 receives 1000 - 400 = 600
        assertEq(leaderCatContract.balanceOf(dex1), 600);

        vm.stopPrank();
    }

}
```

Recommended mitigation To correctly calculate the fees, the formula should be updated to:

```
function transfer(address _to, uint256 _amount)
    external
    notBlacklisted(msg.sender)
    notBlacklisted(_to)
    validAddress(_to)
    noReentrancy
    returns (bool success)
{
    require(_amount > 0, "Transfer amount must be greater than 0");
    require(balanceOf[msg.sender] >= _amount, "Insufficient balance");

    uint256 fee = 0;

    if (isDEX(msg.sender)) {
-       fee = (_amount * buyFee) / MAX_FEE_PERCENTAGE;
+       fee = (_amount * buyFee) / 100;
    } else if (isDEX(_to)) {
-       fee = (_amount * sellFee) / MAX_FEE_PERCENTAGE;
+       fee = (_amount * sellFee) / 100;
    }

    uint256 amountAfterFee = _amount - fee;

    balanceOf[msg.sender] -= _amount;
    balanceOf[_to] += amountAfterFee;

    if (fee > 0) {
        balanceOf[feeRecipient] += fee;
        emit Transfer(msg.sender, feeRecipient, fee);
    }

    emit Transfer(msg.sender, _to, amountAfterFee);
    return true;
}
```

This ensures that the fee is correctly calculated as a percentage of the transaction amount. Additionally, thorough testing should be conducted to verify the correctness of the fee calculations for both buy and sell transactions.

Resolution: Fixed.

5.2 Low

5.2.1 [L-1] Ineffective Timelock for Blacklisting in LEADERCATCONTRACT:updateBlacklist Allows Delayed Response to Malicious Activity

Description

The `LEADERCATCONTRACT:updateBlacklist` function imposes a timelock of 24 hours for blacklisting or unblacklisting addresses. This timelock is applied uniformly, regardless of the address being blacklisted or unblacklisted. The current implementation requires waiting for the timelock to expire before any address can be blacklisted or unblacklisted again.

Impact

The timelock mechanism delays the ability to promptly respond to malicious activities. If a malicious address is identified, the contract owner must wait for the timelock to expire before blacklisting the address, allowing the malicious actor to continue their activities during this period. Similarly, if an address needs to be unblacklisted urgently, the timelock prevents immediate action.

Proof of Concepts

In the test `testBlacklistingAnAddressIsAvailableOnlyEvery24Hours`, the following steps demonstrate the issue: 1. Wait the initial time of 24 hours to blacklist the first user (Alice) 2. Then try to blacklist Bob (this should revert) 3. Wait for another 24 hours and blacklist Bob then (this should pass)

PoC

Place this in a new file under the test folder in a foundry project

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {LEADERCATCONTRACT} from "../src/LeaderContract.sol";

contract LEADERCATCONTRACTTest is Test {
    LEADERCATCONTRACT public leaderCatContract;

    address owner;
    address feeRecipient;

    uint256 timelockDuration = 24 hours;

    function setUp() public {
        feeRecipient = makeAddr("0x123");

        vm.prank(owner);
        leaderCatContract = new LEADERCATCONTRACT(timelockDuration, feeRecipient);
    }

    function testBlacklistingAnAddressIsAvailableOnlyEvery24Hours() public {
        address alice = makeAddr("0x3");
        address bob = makeAddr("0x4");
        // skip the initial 24 hours from the contract deployment
        skip(timelockDuration);
        vm.startPrank(owner);
        leaderCatContract.updateBlacklist(alice, true);

        // since we have set the blacklist for alice we should not be able to set it
            for bob because of the timelock
        // we should get a revert error with the message "Timelock in effect"
        vm.expectRevert("Timelock in effect");
        leaderCatContract.updateBlacklist(bob, true);

        // we skip once again the timelock duration which here should be 24 hours
        skip(24 hours);
        leaderCatContract.updateBlacklist(bob, true);
        vm.stopPrank();
    }

}
```

The contract owner is unable to blacklist a new malicious address (bob) immediately after blacklisting another address (alice), due to the timelock

Recommended mitigation

To address this issue, consider the following mitigations:

1. Remove the timelock for blacklisting operations, allowing the contract owner to blacklist addresses immediately.
2. Implement a separate timelock for unblacklisting operations, if necessary, to prevent abuse.
3. Alternatively, introduce a more granular timelock mechanism that takes into account the specific address being blacklisted or unblacklisted, allowing for immediate action against new malicious addresses while maintaining a timelock for unblacklisting.

Example modification:

```
function updateBlacklist(address account, bool status)
    external
    onlyOwner
-   timelockPassed(this.updateBlacklist.selector)
{
    isBlacklisted[account] = status;
    emit BlacklistUpdated(account, status);
}
```

Resolution: Fixed.

5.3 Informational

5.3.1 [I-1] LEADERCATCONTRACT Does Not Follow ERC20 Standard for Fungible Tokens

Description

The `LEADERCATCONTRACT` is intended to be a token implementation but does not follow the ERC20 standard for fungible tokens. The ERC20 standard defines a set of functions and events that a token contract must implement to ensure compatibility with various wallets, exchanges, and other smart contracts. The current implementation lacks several key functions and events defined by the ERC20 standard, such as transferFrom, approve, and Approval.

Impact

Not following the ERC20 standard can lead to compatibility issues with wallets, exchanges, and other smart contracts that expect a standard ERC20 interface. This can limit the usability and adoption of the token, as it may not be recognized or supported by various platforms and services.

Proof of Concepts

The LEADERCATCONTRACT lacks the following ERC20 standard functions and events: 1. `transferFrom(address from, address to, uint256 value)`: Allows a spender to transfer tokens on behalf of the token owner. 2. `approve(address spender, uint256 value)`: Allows the token owner to approve a spender to transfer tokens on their behalf. 3. `allowance(address owner, address spender)`: Returns the remaining number of tokens that a spender is allowed to transfer on behalf of the token owner. 4. `Approval(address indexed owner, address indexed spender, uint256 value)`: Event emitted when the approve function is called. Recommended mitigation

To ensure compatibility with the ERC20 standard, it is recommended to use the OpenZeppelin ERC20 implementation. OpenZeppelin provides a well-tested and widely-used implementation of the ERC20 standard that includes all the required functions and events.

Resolution: Fixed.

5.3.2 [I-2]: Unnecessary Reentrancy Guard in LEADERCATCONTRACT`::trasnfer`

The `LEADERCATCONTRACT` contract includes a reentrancy guard (noReentrancy modifier) to prevent reentrancy attacks. However, the contract does not make any external calls that could lead to reentrancy vulnerabilities. As a result, the reentrancy guard is unnecessary and adds complexity to the contract without providing any security benefits and costs more gas it uses a few reads and updates on storage memory.

Resolution: Fixed.

5.3.3 [I-3] Unused storage variable LEADERCATCONTRACT`::allowance`

The `LEADERCATCONTRACT` contract includes an allowance mapping, which is typically used in conjunction with a `transferFrom` function to manage token allowances. However, the contract does not implement a `transferFrom` function, making the allowance mapping unused and unnecessary. This adds unnecessary complexity and storage usage to the contract.

Resolution: Fixed.

5.3.4 [I-4]: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, insteaf of `pragma solidity ^0.8.0;` use `pragma solidity 0.8.0`

Resolution: Fixed.

5.3.5 [I-5]: Large literal values multiples of 10000 can be replaced with scientific notation

Use `e` notation, for example: `1e18`, instead of its full numeric value.

1 Found Instances

- Found in src/LeaderContract.sol Line: 7

```
    uint256 public constant totalSupply = 1_000_000_000_000_000_000_000_000; //
         10^24 = 1e24
```

Resolution: Fixed.

5.3.6 [I-6]: Compilation erroe when trying to access isDex mapping

In the `Transfer` function, the isDex mapping values are wrongly retrieved by using () instead of [].

```
 if (isDEX(msg.sender)) {
            fee = (_amount * buyFee) / MAX_FEE_PERCENTAGE;
        } else if (isDEX(_to)) {
            fee = (_amount * sellFee) / MAX_FEE_PERCENTAGE;
        }
```

Resolution: Fixed.