CYBERSEC CONTRACT AUDIT REPORT
SpiderDAO - CTDSEC.COM

Introduction

During January of 2021, SpiderDAO engaged CTDSec to audit smart contracts that they created. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts. SpiderDAO provided CTDSec with access to their code repository and whitepaper.

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bugfree status. The audit documentation is for discussion purposes only.

I always recommend having a bug bounty program opened to detect future bugs.

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the SpiderDAO contract followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code files are considered in-scope for the review:

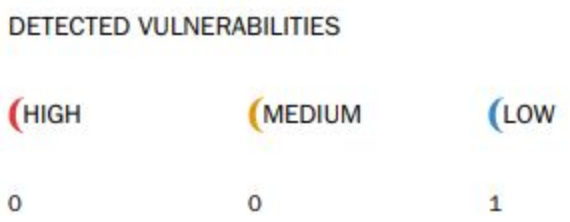
- SpiderDAONest1.sol
- SpiderDAONest12.sol
- SpiderDAONest3.sol
- SpiderDAONest6.sol

Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

- Correctness of the protocol implementation [Result OK]
- User funds are secure on the blockchain and cannot be transferred without user permission [Result OK]
- Vulnerabilities within each component as well as secure interaction between the network components [Result OK]
- Correctly passing requests to the network core [Result OK]
- Data privacy, data leaking, and information integrity [Result OK]
- Susceptible to reentrancy attack [Result OK]
- Key management implementation: secure private key storage and proper management of encryption and signing keys [Result OK]
- Handling large volumes of network traffic [Result OK]
- Resistance to DDoS and similar attacks [Result OK]
- Aligning incentives with the rest of the network [Result OK]
- Any attack that impacts funds, such as draining or manipulating of funds [Result OK]
- Mismanagement of funds via transactions [Result OK]
- Inappropriate permissions and excess authority [Result OK]
- Special token issuance model [Result OK]

Vulnerabilities



Architecture:

Because the resolution of the image does not allow it to be displayed correctly in the document we attach a link:

https://pasteboard.co/JIH0Chx.png

ISSUES

LOW

Multiplication after division.

Solidity operates only with integers. Thus, if the division is done before the multiplication, the rounding errors can increase dramatically.

Multiplication before division may increase the rounding precision.

Issue was solved by the dev team modifying the operators. Now multiplication is done before division.

Locations

```
function _withdrawAfterClose(address from, uint256 amount) private {
    uint256 reward = amount.div(100).mul(REWARD_PERC);
    uint256 payOut = amount.add(reward);
    _stakes[from] = _stakes[from].sub(amount);
    stakedTotal = stakedTotal.sub(amount);
    SpiderDAOToken.safeTransferFrom(rewardAddress, from, payOut);
    emit PaidOut(tokenAddress, from, amount, reward);
}
```

Test cases

As required by the team we made a mock of the erc20 contract in order to test the logic of stake and withdrawal with the provided parameters.

We defined the duration and rewards at the start and after tested the staking of 2 accounts. Also we verified that withdrawal and rewards are correctly applied.

Code:

```
58
59      stakedTotal = (await daoContract.stakedTotal.call()).toNumber();
60      staked = (await daoContract.stakeOf.call(accounts[1])).toNumber();
61      staked2 = (await daoContract.stakeOf.call(accounts[1])).toNumber();
62
63      assert.equal(stakedTotal, 2000, "Incorrect");
64      assert.equal(staked, 1000, "Incorrect");
65      assert.equal(staked2, 1000, "Incorrect");
66
67    });
68
69    it('should withdraw', async () => {
70
71
72      let stakedTotal = (await daoContract.stakedTotal.call()).toNumber();
73      let staked = (await daoContract.stakeOf.call(accounts[1])).toNumber();
74
75      assert.equal(stakedTotal, 0, "Incorrect");
76      assert.equal(staked, 0, "Incorrect");
77
78      await daoContract.stake(1000, { from: accounts[1] });
79      await daoContract.stake(1000, { from: accounts[2] });
80
81      stakedTotal = (await daoContract.stakedTotal.call()).toNumber();
82      staked = (await daoContract.stakeOf.call(accounts[1])).toNumber();
83
84      assert.equal(stakedTotal, 2000, "Incorrect");
85      assert.equal(staked, 1000, "Incorrect");
86
87      await advanceTime(stakeDuration + 1);
88      let balance = (await erc20.balanceOf.call(accounts[1])).toNumber();
89      assert.equal(balance, 100000 - 1000, "Incorrect");
90
91      await daoContract.withdraw(1000, { from: accounts[1] });
92
93      balance = (await erc20.balanceOf.call(accounts[1])).toNumber();
94      assert.equal(balance, 100000 + (1000 * reward / 100), "Incorrect");
95
96      stakedTotal = (await daoContract.stakedTotal.call()).toNumber();
97      staked = (await daoContract.stakeOf.call(accounts[1])).toNumber();
98
99      assert.equal(stakedTotal, 1000, "Incorrect");
100     assert.equal(staked, 0, "Incorrect");
101
102
103   });
104 });
105
```

Testing example:

Provided files to customer:

1. DAO.JS (Include the JS code)
2. spiderdao.zip (Include all test cases)

Summary of the Audit

During the audit the contract was manually reviewed and analyzed with static analysis tools. CTDSEC team have found low security issues and the audit report contains all necessary information related to them.

Low vulnerability issue was solved by the development team applying the measures that are explained in the reported issue.