Smart contract security audit Shardstarter

v.1.0

Table of Contents

1.0 Introduction

1.1 Project engagement

During April of 2023, Shardstarter team 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. Shardstarter provided CTDSec with access to their code repository and whitepaper.

1.2 Disclaimer

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that’s able to withstand the network’s fast-paced and rapidly changing environment, we at CTDSec recommend that Shardstarter team put in place a bug bounty program to encourage further and active analysis of the smart contract.

2.0 Coverage

2.1 Target Code and Revision

For this audit, we performed research, investigation, and review of the Shardstarter 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:

Source file:

https://github.com/Shardstarter/Shardstarter/blob/main/contracts/Staking.sol

2.2 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.

№	Issue description.	Checking status
1	Compiler warnings.	PASSED
2	Race conditions and Reentrancy. Cross-function race conditions.	HIGH ISSUES
3	Possible delays in data delivery.	PASSED
4	Oracle calls.	PASSED
5	Front running.	PASSED
6	Timestamp dependence.	PASSED
7	Integer Overflow and Underflow.	PASSED
8	DoS with Revert.	PASSED
9	DoS with block gas limit.	PASSED
10	Methods execution permissions.	PASSED
11	Economy model. If application logic is based on an incorrect economic model, the application would not function correctly and participants would incur financial losses. This type of issue is most often found in bonus rewards systems, Staking and Farming contracts, Vault and Vesting contracts, etc.	PASSED
12	The impact of the exchange rate on the logic.	PASSED
13	Private user data leaks.	PASSED
14	Malicious Event log.	PASSED
15	Scoping and Declarations.	PASSED
16	Uninitialized storage pointers.	PASSED

17	Arithmetic accuracy.	PASSED
18	Design Logic.	PASSED
19	Cross-function race conditions.	PASSED
20	Safe Zeppelin module.	PASSED
21	Fallback function security.	PASSED
22	Overpowered functions / Owner privileges	PASSED

3.0 Security Issues

3.1 High severity issues [1]

1. Re-entrancy attack

Function: ***Stake()***

Problem: This function is vulnerable to re-entrancy attacks.

Fix: Add the nonReentrant modifier to the function, which will prevent multiple calls to the same function before the previous one has completed.

3.2 Medium severity issues [0]

No medium severity issues found.

3.3 Low severity issues [4]

1. Main Wallet Address Initialization

Function: ***constructor()***

Problem: If the access to the mainWallet is lost, setting the contract address in this way is not recommended.

Fix: It is highly recommended to set the mainWallet via arguments during the contract deployment process, and then add a setter function that allows the owner of the contract to update the mainWallet address as needed.

2. StakingToken Balance Check

Function: ***Stake()***

Problem: The function needs to check whether the amount sent by msg.sender is greater than the balance of the StakingToken held by msg.sender.

Fix: Add the following line of code: require(_amount <= IERC20(stakingToken).balanceOf(msg.sender), "Insufficient balance");

3. Balance Check for Withdrawal

Function: ***Withdraw()***

Problem: The function needs to verify that the withdrawal amount requested by msg.sender is not greater than their available balance.

Fix: Add the following line of code: require(balances[msg.sender] >= _amount, "Insufficient balance");

4. Reward Balance Check

Function: ***getReward()***

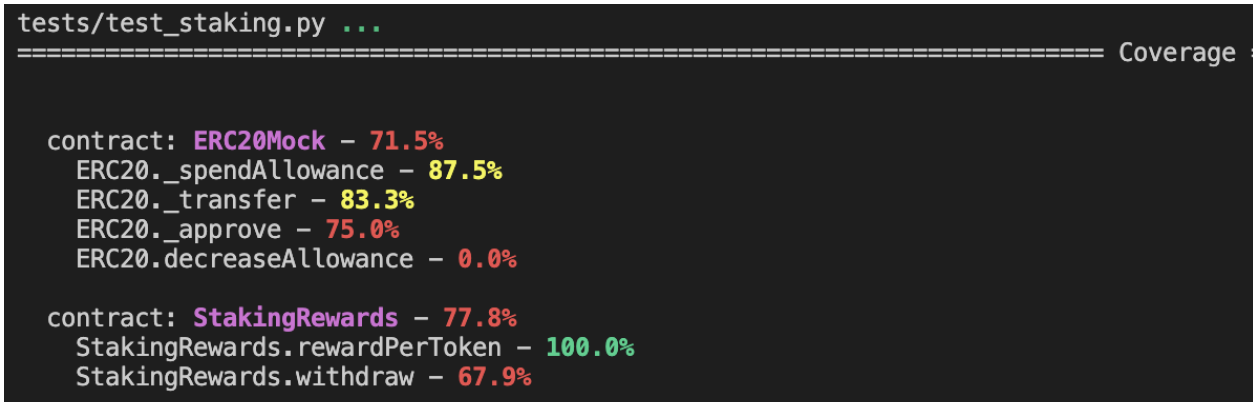
Problem: The function needs to verify that the reward balance held by msg.sender is greater than zero before any withdrawal can occur.

Fix: Add the following line of code: require(reward > 0, "No reward to withdraw");

4.0 Testing coverage - python

During the testing phase, custom use cases were written to cover all the logic of contracts in python language. <u>**Check "5 Annexes" to see the testing code.*</u>

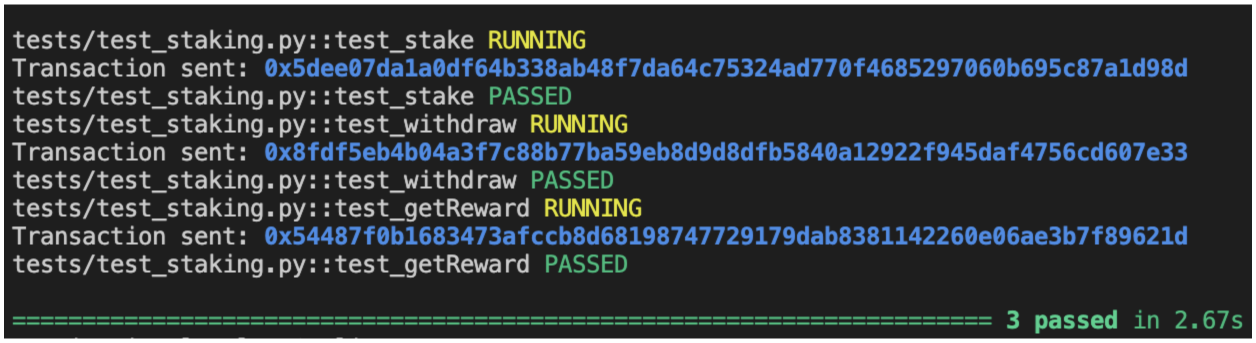
<u>Shardstarter tests</u>



```
tests/test_staking.py ...
========================================================================== Coverage


  contract: ERC20Mock - 71.5%
    ERC20._spendAllowance - 87.5%
    ERC20._transfer - 83.3%
    ERC20._approve - 75.0%
    ERC20.decreaseAllowance - 0.0%

  contract: StakingRewards - 77.8%
    StakingRewards.rewardPerToken - 100.0%
    StakingRewards.withdraw - 67.9%
```



```
tests/test_staking.py::test_stake RUNNING
Transaction sent: 0x5dee07da1a0df64b338ab48f7da64c75324ad770f4685297060b695c87a1d98d
tests/test_staking.py::test_stake PASSED
tests/test_staking.py::test_withdraw RUNNING
Transaction sent: 0x8fdf5eb4b04a3f7c88b77ba59eb8d9d8dfb5840a12922f945daf4756cd607e33
tests/test_staking.py::test_withdraw PASSED
tests/test_staking.py::test_getReward RUNNING
Transaction sent: 0x54487f0b1683473afccb8d68198747729179dab8381142260e06ae3b7f89621d
tests/test_staking.py::test_getReward PASSED

==================================================================== 3 passed in 2.67s
```

5.0 Annexes

Staking testing:

```
from brownie import (
    reverts,
)

from scripts.helpful_scripts import (
    get_account,
    evm_increase_time,
    ONE_ETH,
)

from scripts.deploy import (
    deploy_staking_rewards
)

def test_stake(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    staking_re, staking_token, rewards_token =
deploy_staking_rewards(owner)

    staking_token.approve(staking_re.address, 5e18, {"from": owner})
    # asserts
    tx = staking_re.stake(ONE_ETH, {"from": owner})
    assert tx.events["Transfer"] is not None
    assert tx.events["Transfer"]["from"] == owner
    assert tx.events["Transfer"]["to"] == staking_re.address
    assert tx.events["Transfer"]["value"] == ONE_ETH
    assert staking_re.balances(owner) == ONE_ETH
    assert staking_re._totalSupply() == ONE_ETH
    assert staking_re.earned(owner) == 0
    assert staking_re.rewards(owner) == 0
    assert staking_re.rewardPerToken() == 0

    # Increase de chain timestamp one day (seconds)
    evm_increase_time(86400)
    staking_re.stake(ONE_ETH, {"from": owner})
    assert staking_re.balances(owner) == ONE_ETH * 2
    assert staking_re._totalSupply() == ONE_ETH * 2
    assert staking_re.earned(owner) >= 86400
    assert staking_re.rewards(owner) >= 86400
    assert staking_re.rewardPerToken() >= 86400

    # Stake with a different account
    with reverts():
        staking_re.stake(ONE_ETH, {"from": other}) # not allowance and not
token staking tokens
    # transfer staking tokens and add allowance
    staking_token.transfer(other, ONE_ETH, {"from": owner})
    staking_token.approve(staking_re.address, 5e18, {"from": other})
    assert staking_re.balances(other) == 0
    staking_re.stake(ONE_ETH, {"from": other})
    assert staking_re.balances(other) == ONE_ETH
    assert staking_re._totalSupply() == ONE_ETH * 3
    assert staking_re.earned(other) == 0
    assert staking_re.rewards(other) == 0

def test_withdraw(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    extra = get_account(2)
    staking_re, staking_token, rewards_token =
deploy_staking_rewards(owner)

    staking_token.transfer(other, ONE_ETH, {"from": owner})
    staking_token.approve(staking_re.address, ONE_ETH * 5, {"from": other})
    staking_token.approve(staking_re.address, ONE_ETH * 5, {"from": owner})

    #stake some tokens to withdraw later with differents addresses
    staking_re.stake(ONE_ETH, {"from": owner})
    staking_re.stake(ONE_ETH, {"from": other})

    # asserts
    with reverts():
        staking_re.withdraw(ONE_ETH, {"from": extra}) # not enough amount
to withdraw
```

```
    assert staking_re.balances(other) == ONE_ETH
    assert staking_re._totalSupply() == ONE_ETH * 2
    # withdraw paying fees
    tx = staking_re.withdraw(ONE_ETH, {"from": other})
    assert tx.events["Transfer"] is not None
    assert len(tx.events["Transfer"]) == 2
    assert staking_re.balances(other) == 0
    assert staking_re._totalSupply() == ONE_ETH
    assert staking_token.balanceOf(other) == 750000000000000000
    # withdraw with no fees after locktime
    evm_increase_time(86400 * 60)
    assert staking_re.balances(owner) == ONE_ETH
    assert staking_re._totalSupply() == ONE_ETH
    # withdraw paying fees
    tx = staking_re.withdraw(ONE_ETH, {"from": owner})
    assert tx.events["Transfer"] is not None
    assert len(tx.events["Transfer"]) == 1
    assert staking_re.balances(owner) == 0
    assert staking_re._totalSupply() == 0

def test_getReward(only_local):
    # arrage
    owner = get_account(0)
    other = get_account(1)
    staking_re, staking_token, rewards_token =
deploy_staking_rewards(owner)

    staking_token.transfer(other, ONE_ETH * 2, {"from": owner})
    staking_token.approve(staking_re.address, ONE_ETH * 5, {"from": other})
    staking_token.approve(staking_re.address, ONE_ETH * 5, {"from": owner})

    #stake some tokens to get reward later
    staking_re.stake(ONE_ETH, {"from": owner})
    evm_increase_time(86400)
    staking_re.stake(ONE_ETH, {"from": owner})

    # asserts
    with reverts():
        staking_re.getReward({"from": owner}) # not enought balance in
rewards token

    # transfer some amount to staking contract address
    rewards_token.transfer(staking_re.address, ONE_ETH, {"from": owner})
    assert staking_re.rewards(owner) == 86400
    staking_re.getReward({"from": owner})
    assert staking_re.rewards(owner) == 0
    #stake some tokens to get reward later from another account
    staking_re.stake(ONE_ETH, {"from": other})
    evm_increase_time(86400)
    staking_re.stake(ONE_ETH, {"from": other})
    assert staking_re.rewards(other) == 28800
    staking_re.getReward({"from": other})
    assert staking_re.rewards(other) == 0
```

6.0 Summary of the audit

High vulnerabilities found that need to be fixed before deploying the contracts.