The Solution to remove all vulnerabilities and improve all Functions with Security

- Ownable.sol

```
contract Ownable {
    address public owner;

    event OwnershipTransferred(address indexed previousOwner, address

    indexed newOwner);

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Ownable: caller is not the owner");
        _;
    }

    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(owner, address(0));
        owner = address(0);
    }
    function transferOwnership(address newOwner) public onlyOwner {
        _transferOwnership(newOwner);
    }
    function _transferOwnership(address newOwner) internal {
        require(newOwner != address(0), "Ownable: new owner is the zero
address");
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }
}
```

- IERC20.sol

```
interface IERC20 {
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address recipient, uint256 amount) external returns (bool);
    function allowance(address owner, address spender) external view returns
(uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(address sender, address recipient, uint256 amount)
external returns (bool);
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256
value);
}
```

- StakedWrapper.sol

```
contract StakedWrapper {
    uint256 public totalSupply;
    uint128 public buyback = 2; //defined in percentage
    mapping(address => uint256) private _balances;
    IERC20 public stakedToken;

    event Staked(address indexed user, uint256 amount);
    event Withdrawn(address indexed user, uint256 amount);

    address public beneficiary =
address(0xDbfd6dAbD2Eaf53e5dBDc5E96fFB7E5E6B201F69);

    function balanceOf(address account) public view returns (uint256) {
        return _balances[account];
    }

    string constant _transferErrorMessage = "staked token transfer failed";

    function stakeFor(address forWhom, uint128 amount) public payable virtual {
        IERC20 st = stakedToken;
        if(st == IERC20(address(0))) { //eth
            unchecked {
                totalSupply += msg.value;
                _balances[forWhom] += msg.value;
            }
        } else {
            require(msg.value == 0, "Zero Eth not allowed");
            require(amount > 0, "Stake should be greater than zero");
            require(st.transferFrom(msg.sender, address(this), amount),
_transferErrorMessage);
            unchecked {
                totalSupply += amount;
                _balances[forWhom] += amount;
            }
        }

        emit Staked(forWhom, amount);
    }

    function withdraw(uint128 amount) public virtual {
        require(amount <= _balances[msg.sender], "withdraw: balance is lower");
        unchecked {
            _balances[msg.sender] -= amount;
            totalSupply = totalSupply-amount;
        }
        IERC20 st = stakedToken;
        if(st == IERC20(address(0))) { //eth
            uint128 val = (amount*buyback)/100;
            payable(beneficiary).transfer(val);
            // beneficiary.call{value: val}("");
            (bool success_, ) = msg.sender.call{value: amount-val}("");
            require(success_, "eth transfer failure");
        }
        else {
            require(stakedToken.transfer(msg.sender, amount),
            _transferErrorMessage);
        }
        emit Withdrawn(msg.sender, amount);
    }
}
```

- **RewardsETH.sol**

```
contract RewardsETH is StakedWrapper, Ownable {
    IERC20 public rewardToken;
    uint256 public rewardRate;
    uint64 public periodFinish;
    uint64 public lastUpdateTime;
    uint128 public rewardPerTokenStored;

    struct UserRewards {
        uint128 userRewardPerTokenPaid;
        uint128 rewards;
    }
    mapping(address => UserRewards) public userRewards;

    event RewardAdded(uint256 reward);
    event RewardPaid(address indexed user, uint256 reward);

    uint256 public maxStakingAmount = 2 * 10**0 * 10**17; //0.2 ETH

    constructor(IERC20 _rewardToken, IERC20 _stakedToken) {
        rewardToken = _rewardToken;
        stakedToken = _stakedToken;
    }

    modifier updateReward(address account) {
        uint128 _rewardPerTokenStored = rewardPerToken();
        lastUpdateTime = lastTimeRewardApplicable();
        rewardPerTokenStored = _rewardPerTokenStored;
        userRewards[account].rewards = earned(account);
        userRewards[account].userRewardPerTokenPaid = _rewardPerTokenStored;
        _;
    }
    function lastTimeRewardApplicable() public view returns (uint64) {
        uint64 blockTimestamp = uint64(block.timestamp);
        return blockTimestamp < periodFinish ? blockTimestamp : periodFinish;
    }
    function rewardPerToken() public view returns (uint128) {
        uint256 totalStakedSupply = totalSupply;
        if (totalStakedSupply == 0) {
            return rewardPerTokenStored;
        }
        unchecked {
            uint256 rewardDuration = lastTimeRewardApplicable()-lastUpdateTime;
            return uint128(rewardPerTokenStored +
rewardDuration*rewardRate*1e18/totalStakedSupply);
        }
    }
    function earned(address account) public view returns (uint128) {
        unchecked {
            return uint128(balanceOf(account)*(rewardPerToken()-
userRewards[account].userRewardPerTokenPaid)/1e18 +
userRewards[account].rewards);
        }
    }
    function stake(uint128 amount) external payable {
        require(amount < maxStakingAmount, "amount exceed max staking amount");
        stakeFor(msg.sender, amount);
    }
    function stakeFor(address forWhom, uint128 amount) public payable override
updateReward(forWhom) {
        super.stakeFor(forWhom, amount);
    }
    function withdraw(uint128 amount) public override updateReward(msg.sender) {
        super.withdraw(amount);
    }
    function exit() external {
        getReward();
        withdraw(uint128(balanceOf(msg.sender)));
    }
    function getReward() public updateReward(msg.sender) {
        uint256 reward = earned(msg.sender);
        if (reward > 0) {
            userRewards[msg.sender].rewards = 0;
            require(rewardToken.transfer(msg.sender, reward), "reward transfer
failed");
            emit RewardPaid(msg.sender, reward);
        }
    }
    function setRewardParams(uint128 reward, uint64 duration) external onlyOwner
{
        unchecked {
            require(reward > 0);
            rewardPerTokenStored = rewardPerToken();
            uint64 blockTimestamp = uint64(block.timestamp);
            uint256 maxRewardSupply = rewardToken.balanceOf(address(this));
            if(rewardToken == stakedToken)
                maxRewardSupply -= totalSupply;
            uint256 leftover = 0;
            if (blockTimestamp >= periodFinish) {
                rewardRate = reward/duration;
            } else {
                uint256 remaining = periodFinish-blockTimestamp;
                leftover = remaining*rewardRate;
                rewardRate = (reward+leftover)/duration;
            }
            require(reward+leftover <= maxRewardSupply, "not enough tokens");
            lastUpdateTime = blockTimestamp;
            periodFinish = blockTimestamp+duration;
            emit RewardAdded(reward);
        }
    }
    function withdrawReward() external onlyOwner {
        uint256 rewardSupply = rewardToken.balanceOf(address(this));
        //ensure funds staked by users can't be transferred out
        if(rewardToken == stakedToken)
            rewardSupply -= totalSupply;
        require(rewardToken.transfer(msg.sender, rewardSupply));
        rewardRate = 0;
        periodFinish = uint64(block.timestamp);
    }

    function setMaxStakingAmount(uint256 value) external onlyOwner {
        require(value > 0);
        maxStakingAmount = value;
    }
    function setBuyback(uint128 value) external onlyOwner {
        buyback = value;
    }
    function setBuyBackAddr(address addr) external onlyOwner {
        beneficiary = addr;
    }
}
```

- Elements (Variables and Functions)

There are 6 variables, 1 mapping data, 1 struct data, and 2 events.

All variables are declared as public variables.

There are 1 modifier, 3 view functions, and 10 public and external functions in this contract.

- vulnerabilities

The first major vulnerability I can find is that all variables and functions were public.

As we all know, smart contracts are the core of blockchain-related projects, and all transactions are processed and stored through smart contracts.

So we have some things all users can see and some things they shouldn't.

So that Importantly, this contract has been made vulnerable to attack.

Hacker can reentrancy attack to this contract because this contract didn't have the protect function from attack. (Withdraw() function, exit() function, withdrawReward() function)

The Reentrancy attack is one of the most destructive attacks in the Solidity smart contract. A reentrancy attack occurs when a function makes an external call to another untrusted contract. Then the untrusted contract makes a recursive call back to the original function in an attempt to drain funds.

When the contract fails to update its state before sending funds, the attacker can continuously call the withdraw function to drain the contract's funds.

A famous real-world Reentrancy attack is the DAO attack which caused a loss of 60 million US dollars.

Then next importantly, Smart contracts are self-executing codes that cannot be modified once uploaded. Often, these codes are poorly-written, thus featuring bugs that make them vulnerable to attacks. These bugs can trigger unintended tasks that can result in tremendous losses for investors.

So we should be able to edit smart contract after deploy.

- Solutions

I changed the function visibilities with internal, private, public, external.

I implemented the Reentrancy Guard library.

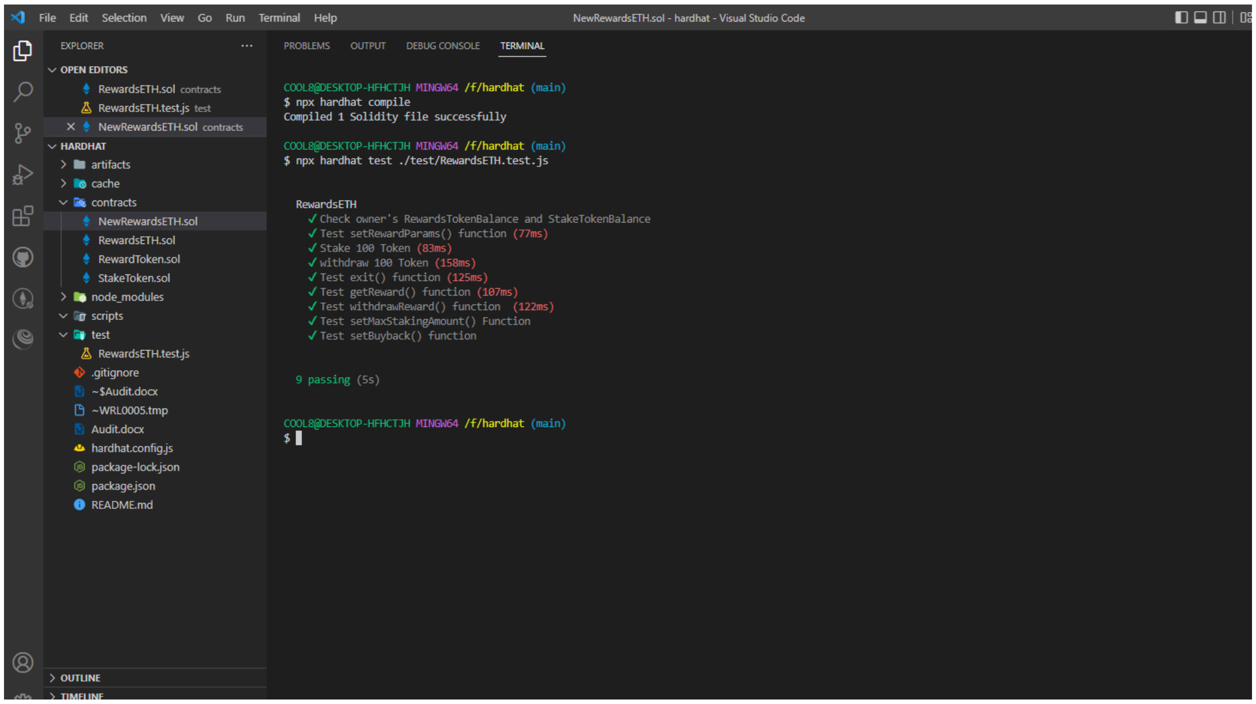
A successful reentrancy attack can be devastating and possibly drain all the funds in the victim's contract, so it is important to be aware of potential vulnerabilities and implement effective safeguards. The CEI pattern should be implemented by default, whether there is a vulnerability or not; it's simply good practice. Additional security can be accomplished through the use of reentrancy guards and/or pull payments.

- Conclusion

I built this smart contract with hardhat.

I wrote a unit test script to check that all functions work well.

I tested all the functions with my unit test and everything worked fine.



```
COOL8@DESKTOP-HFHCTJH MINGW64 /f/hardhat (main)
$ npx hardhat compile
Compiled 1 Solidity file successfully

COOL8@DESKTOP-HFHCTJH MINGW64 /f/hardhat (main)
$ npx hardhat test ./test/RewardsETH.test.js


  RewardsETH
    ✓ Check owner's RewardsTokenBalance and StakeTokenBalance
    ✓ Test setRewardParams() function (77ms)
    ✓ Stake 100 Token (83ms)
    ✓ withdraw 100 Token (158ms)
    ✓ Test exit() function (125ms)
    ✓ Test getReward() function (107ms)
    ✓ Test withdrawReward() function  (122ms)
    ✓ Test setMaxStakingAmount() Function
    ✓ Test setBuyback() function


  9 passing (5s)


COOL8@DESKTOP-HFHCTJH MINGW64 /f/hardhat (main)
$ 
```