1. Contracts list: the contracts need to audit

a	UP Token	UpToken.sol
b	UP Farm	UpFarm.sol
c	Base Strategy	Strategy.sol
d	Strategy for PCS	StrategyPCS.sol
e	Strategy for marsecosystem	StrategyMars.sol
f	stakingRewars	StakingRewars.sol
g	vesting	VestingMaster.sol

2. Changes of my contract

UpFarm: UPfarm forked the autofarmV2 and add a vesting strategy. The red is the related functions.

```
// SPDX-License-Identifier: MIT

pragma solidity 0.6.12;

import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";

import "./interfaces/IVestingMaster.sol";
import "./interfaces/IStrategy.sol";

interface IUPToken {
    function mint(address _to, uint256 _amount) external;
}

contract UpFarm is Ownable, ReentrancyGuard {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    struct UserInfo {
        uint256 shares;
        uint256 rewardDebt;
    }

    struct PoolInfo {
        IERC20 want;
        uint256 allocPoint;
        uint256 lastRewardBlock;
        uint256 accUPPerShare;
        address strat;
```

```
    }

    address public UP;
    address public vestingMaster;
    uint256 public UPPerBlock;
    uint256 public startBlock;

    PoolInfo[] public poolInfo;
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;
    uint256 public totalAllocPoint = 0;

    event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
    event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
    event EmergencyWithdraw(
        address indexed user,
        uint256 indexed pid,
        uint256 amount
    );
    event UpdateUPPerBlock(address indexed user,uint256 amount);
    event SetVestingMaster(address indexed user,address vesting);

    constructor(
        address _upAddress,
        address _vestingMaster,
        uint256 _upPerBlock,
        uint256 _startBlock
    ) public {
        UP = _upAddress;
        vestingMaster = _vestingMaster;
        UPPerBlock = _upPerBlock;
        startBlock = _startBlock;
    }

    function poolLength() public view returns (uint256) {
        return poolInfo.length;
    }

    function add(
        uint256 _allocPoint,
        IERC20 _want,
        bool _withUpdate,
        address _strat
    ) public onlyOwner {
        if (_withUpdate) {
```

```
            massUpdatePools();
        }
        uint256 lastRewardBlock =
            block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo({
                want: _want,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accUPPerShare: 0,
                strat: _strat
            })
        );
    }

    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    function getMultiplier(uint256 _from, uint256 _to) public pure returns (uint256) {
        return _to.sub(_from);
    }

    function pendingUP(uint256 _pid, address _user) public view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accUPPerShare = pool.accUPPerShare;
        uint256 sharesTotal = IStrategy(pool.strat).sharesTotal();
        if (block.number > pool.lastRewardBlock && sharesTotal != 0) {
            uint256 multiplier =
                getMultiplier(pool.lastRewardBlock, block.number);
            uint256 UPReward =
                multiplier.mul(UPPerBlock).mul(pool.allocPoint).div(
```

```
                totalAllocPoint
            );
        accUPPerShare = accUPPerShare.add(
            UPReward.mul(1e12).div(sharesTotal)
        );
    }
    return user.shares.mul(accUPPerShare).div(1e12).sub(user.rewardDebt);
}

function stakedWantTokens(uint256 _pid, address _user) public view returns (uint256) {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][_user];

    uint256 sharesTotal = IStrategy(pool.strat).sharesTotal();
    uint256 wantLockedTotal =
        IStrategy(poolInfo[_pid].strat).wantLockedTotal();
    if (sharesTotal == 0) {
        return 0;
    }
    return user.shares.mul(wantLockedTotal).div(sharesTotal);
}

function massUpdatePools() public {
    for (uint256 pid = 0; pid < poolInfo.length; ++pid) {
        updatePool(pid);
    }
}

function updatePool(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    if (block.number <= pool.lastRewardBlock) {
        return;
    }
    uint256 sharesTotal = IStrategy(pool.strat).sharesTotal();
    if (sharesTotal == 0) {
        pool.lastRewardBlock = block.number;
        return;
    }
    uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
    if (multiplier == 0) {
        return;
    }
    uint256 UPReward =
```

```
                multiplier.mul(UPPerBlock).mul(pool.allocPoint).div(
                    totalAllocPoint
                );
            pool.accUPPerShare = pool.accUPPerShare.add(
                UPReward.mul(1e12).div(sharesTotal)
            );
            pool.lastRewardBlock = block.number;
        }

        function deposit(uint256 _pid, uint256 _wantAmt) public nonReentrant {
            updatePool(_pid);
            PoolInfo storage pool = poolInfo[_pid];
            UserInfo storage user = userInfo[_pid][msg.sender];

            if (user.shares > 0) {
                uint256 pending =
                    user.shares.mul(pool.accUPPerShare).div(1e12).sub(
                        user.rewardDebt
                    );
                if (pending > 0) {
                    uint256 locked;
                    if (vestingMaster != address(0)) {
                        locked = pending
                            .div(IVestingMaster(vestingMaster).lockedPeriodAmount()
.add(1))
                            .mul(IVestingMaster(vestingMaster).lockedPeriodAmount(
));
                    }
                    safeUPTransfer(msg.sender, pending.sub(locked));
                    if (locked > 0) {
                        uint256 actualAmount = safeUPTransfer(
                            vestingMaster,
                            locked
                        );
                        IVestingMaster(vestingMaster).lock(msg.sender,
actualAmount);
                    }
                }
            }
            if (_wantAmt > 0) {
                pool.want.safeTransferFrom(
                    address(msg.sender),
                    address(this),
                    _wantAmt
```

```
                );
                pool.want.safeIncreaseAllowance(pool.strat, _wantAmt);
                uint256 sharesAdded =
                    IStrategy(poolInfo[_pid].strat).deposit(_wantAmt);
                user.shares = user.shares.add(sharesAdded);
            }
            user.rewardDebt = user.shares.mul(pool.accUPPerShare).div(1e12);
            emit Deposit(msg.sender, _pid, _wantAmt);
        }

        function withdraw(uint256 _pid, uint256 _wantAmt) public nonReentrant {
            updatePool(_pid);

            PoolInfo storage pool = poolInfo[_pid];
            UserInfo storage user = userInfo[_pid][msg.sender];

            uint256 wantLockedTotal =
                IStrategy(poolInfo[_pid].strat).wantLockedTotal();
            uint256 sharesTotal = IStrategy(poolInfo[_pid].strat).sharesTotal();

            require(user.shares > 0, "user.shares is 0");
            require(sharesTotal > 0, "sharesTotal is 0");

            uint256 pending =
                user.shares.mul(pool.accUPPerShare).div(1e12).sub(
                    user.rewardDebt
                );
            if (pending > 0) {
                uint256 locked;
                if (vestingMaster != address(0)) {
                    locked = pending
                        .div(IVestingMaster(vestingMaster).lockedPeriodAmount().add
(1))
                        .mul(IVestingMaster(vestingMaster).lockedPeriodAmount());
                }
                safeUPTransfer(msg.sender, pending.sub(locked));
                if (locked > 0) {
                    uint256 actualAmount = safeUPTransfer(
                        vestingMaster,
                        locked
                    );
                    IVestingMaster(vestingMaster).lock(msg.sender, actualAmount);
                }
            }
```

```
        uint256 amount = user.shares.mul(wantLockedTotal).div(sharesTotal);
        if (_wantAmt > amount) {
            _wantAmt = amount;
        }
        if (_wantAmt > 0) {
            uint256 sharesRemoved =
                IStrategy(poolInfo[_pid].strat).withdraw(_wantAmt);

            if (sharesRemoved > user.shares) {
                user.shares = 0;
            } else {
                user.shares = user.shares.sub(sharesRemoved);
            }

            uint256 wantBal = IERC20(pool.want).balanceOf(address(this));
            if (wantBal < _wantAmt) {
                _wantAmt = wantBal;
            }
            pool.want.safeTransfer(address(msg.sender), _wantAmt);
        }
        user.rewardDebt = user.shares.mul(pool.accUPPerShare).div(1e12);
        emit Withdraw(msg.sender, _pid, _wantAmt);
    }

    function withdrawAll(uint256 _pid) public {
        withdraw(_pid, uint256(-1));
    }

    function emergencyWithdraw(uint256 _pid) public nonReentrant {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        uint256 wantLockedTotal =
            IStrategy(pool.strat).wantLockedTotal();
        uint256 sharesTotal = IStrategy(pool.strat).sharesTotal();
        uint256 amount = user.shares.mul(wantLockedTotal).div(sharesTotal);

        IStrategy(poolInfo[_pid].strat).withdraw(amount);
        uint256 wantBal = IERC20(pool.want).balanceOf(address(this));
        if (wantBal < amount) {
            amount = wantBal;
        }
        pool.want.safeTransfer(msg.sender, amount);
```

```
        emit EmergencyWithdraw(msg.sender, _pid, amount);
        user.shares = 0;
        user.rewardDebt = 0;
    }

    function safeUPTransfer(address _to, uint256 _UPAmt) internal returns(uint256) {
        uint256 UPBal = IERC20(UP).balanceOf(address(this));
        if (_UPAmt > UPBal) {
            _UPAmt = UPBal;
        }

        IERC20(UP).safeTransfer(_to, _UPAmt);
        return _UPAmt;
    }

    function setVestingMaster(address _master) public onlyOwner {
        vestingMaster = _master;
        emit SetVestingMaster(msg.sender,_master);
    }

    function updateUPPerBlock(uint256 _upPerBlock) public onlyOwner {
        UPPerBlock = _upPerBlock;
        emit UpdateUPPerBlock(msg.sender,_upPerBlock);
    }

    function inCaseTokensGetStuck(address _token, uint256 _amount) public
onlyOwner {
        require(_token != UP, "!safe");
        IERC20(_token).safeTransfer(msg.sender, _amount);
    }
}
```

Strategy: forked form the AutoFarm StatX2, the red is the collect strategy to get all the profile.

```
// SPDX-License-Identifier: MIT

pragma solidity 0.6.12;

import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Address.sol";
```

```
import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
import "@openzeppelin/contracts/utils/Pausable.sol";

import "./interfaces/IPancakeswapFarm.sol";
import "./interfaces/IPancakeRouter02.sol";
import "./interfaces/IWBNB.sol";

abstract contract Strategy is Ownable, ReentrancyGuard, Pausable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    bool public isCAKEStaking;
    bool public isSameAssetDeposit;
    bool public isAutoComp;
    bool public isCollect;

    address public farmContractAddress;
    uint256 public pid;
    address public wantAddress;
    address public token0Address;
    address public token1Address;
    address public earnedAddress;
    address public uniRouterAddress;

    address public wbnbAddress;
    address public UPFarmAddress;
    address public UPAddress;
    address public govAddress;
    bool public onlyGov = false;

    uint256 public lastEarnBlock = 0;
    uint256 public wantLockedTotal = 0;
    uint256 public sharesTotal = 0;

    uint256 public controllerFee = 0; // 70;
    uint256 public constant controllerFeeMax = 10000; // 100 = 1%
    uint256 public constant controllerFeeUL = 300;

    uint256 public buyBackRate = 0; // 250;
    uint256 public constant buyBackRateMax = 10000; // 100 = 1%
    uint256 public constant buyBackRateUL = 800;
    address public buyBackAddress =
0x000000000000000000000000000000000000dEaD;
    address public rewardsAddress;
```

```
    uint256 public entranceFeeFactor = 9990; // < 0.1% entrance fee - goes to pool + prevents front-running
    uint256 public constant entranceFeeFactorMax = 10000;
    uint256 public constant entranceFeeFactorLL = 9950; // 0.5% is the max entrance fee settable. LL = lowerlimit

    uint256 public withdrawFeeFactor = 10000; // 0.1% withdraw fee - goes to pool
    uint256 public constant withdrawFeeFactorMax = 10000;
    uint256 public constant withdrawFeeFactorLL = 9950; // 0.5% is the max entrance fee settable. LL = lowerlimit

    uint256 public slippageFactor = 950; // 5% default slippage tolerance
    uint256 public constant slippageFactorUL = 995;

    address[] public earnedToUpPath;
    address[] public earnedToToken0Path;
    address[] public earnedToToken1Path;
    address[] public token0ToEarnedPath;
    address[] public token1ToEarnedPath;

    event SetSettings(
        uint256 _entranceFeeFactor,
        uint256 _withdrawFeeFactor,
        uint256 _controllerFee,
        uint256 _buyBackRate,
        uint256 _slippageFactor
    );

    event SetGov(address _govAddress);
    event SetOnlyGov(bool _onlyGov);
    event SetUniRouterAddress(address _uniRouterAddress);
    event SetBuyBackAddress(address _buyBackAddress);
    event SetRewardsAddress(address _rewardsAddress);

    modifier onlyAllowGov() {
        require(msg.sender == govAddress, "!gov");
        _;
    }

    function deposit(uint256 _wantAmt)
        public
        virtual
        onlyOwner
```

```
        nonReentrant
        whenNotPaused
        returns (uint256)
    {
        IERC20(wantAddress).safeTransferFrom(
            address(msg.sender),
            address(this),
            _wantAmt
        );

        uint256 sharesAdded = _wantAmt;
        if (wantLockedTotal > 0 && sharesTotal > 0) {
            sharesAdded = _wantAmt
                .mul(sharesTotal)
                .mul(entranceFeeFactor)
                .div(wantLockedTotal)
                .div(entranceFeeFactorMax);
        }
        sharesTotal = sharesTotal.add(sharesAdded);

        if (isAutoComp) {
            _farm();
        } else {
            wantLockedTotal = wantLockedTotal.add(_wantAmt);
        }

        return sharesAdded;
    }

    function farm() public virtual nonReentrant {
        _farm();
    }

    function _farm() internal virtual {
        require(isAutoComp, "!isAutoComp");
        uint256 wantAmt = IERC20(wantAddress).balanceOf(address(this));
        wantLockedTotal = wantLockedTotal.add(wantAmt);
        IERC20(wantAddress).safeIncreaseAllowance(farmContractAddress,
wantAmt);

        if (isCAKEStaking) {
            IPancakeswapFarm(farmContractAddress).enterStaking(wantAmt);
        } else {
            IPancakeswapFarm(farmContractAddress).deposit(pid, wantAmt);
```

```
        }
    }

    function _unfarm(uint256 _wantAmt) internal virtual {
        if (isCAKEStaking) {
            IPancakeswapFarm(farmContractAddress).leaveStaking(_wantAmt);
        } else {
            IPancakeswapFarm(farmContractAddress).withdraw(pid, _wantAmt);
        }
    }

    function _collect() internal virtual {
        if (earnedAddress == wbnbAddress) {
            _wrapBNB();
        }
        uint256 earnedAmt = IERC20(earnedAddress).balanceOf(address(this));

        if (earnedAmt > 0 && rewardsAddress != address(0)) {
            IERC20(earnedAddress).safeTransfer(rewardsAddress, earnedAmt);
        }
    }

    function withdraw( uint256 _wantAmt)
        public
        virtual
        onlyOwner
        nonReentrant
        returns (uint256)
    {
        require(_wantAmt > 0, "_wantAmt <= 0");

        uint256 sharesRemoved = _wantAmt.mul(sharesTotal).div(wantLockedTotal);
        if (sharesRemoved > sharesTotal) {
            sharesRemoved = sharesTotal;
        }
        sharesTotal = sharesTotal.sub(sharesRemoved);

        if (withdrawFeeFactor < withdrawFeeFactorMax) {
            _wantAmt = _wantAmt.mul(withdrawFeeFactor).div(
                withdrawFeeFactorMax
            );
        }

        if (isAutoComp) {
```

```
            _unfarm(_wantAmt);
        }

        uint256 wantAmt = IERC20(wantAddress).balanceOf(address(this));
        if (_wantAmt > wantAmt) {
            _wantAmt = wantAmt;
        }

        if (wantLockedTotal < _wantAmt) {
            _wantAmt = wantLockedTotal;
        }

        wantLockedTotal = wantLockedTotal.sub(_wantAmt);

        IERC20(wantAddress).safeTransfer(owner(), _wantAmt);

        return sharesRemoved;
    }

    function harvest() internal virtual {
        _unfarm(0);
    }

    function earn() public virtual nonReentrant whenNotPaused {
        require(isAutoComp, "!isAutoComp");
        if (onlyGov) {
            require(msg.sender == govAddress, "!gov");
        }

        harvest();

        if (isCollect) {
            _collect();
            lastEarnBlock = block.number;
            return;
        }

        if (earnedAddress == wbnbAddress) {
            _wrapBNB();
        }

        uint256 earnedAmt = IERC20(earnedAddress).balanceOf(address(this));

        earnedAmt = distributeFees(earnedAmt);
```

```
        earnedAmt = buyBack(earnedAmt);

        if (isCAKEStaking || isSameAssetDeposit) {
            lastEarnBlock = block.number;
            _farm();
            return;
        }

        IERC20(earnedAddress).safeApprove(uniRouterAddress, 0);
        IERC20(earnedAddress).safeIncreaseAllowance(
            uniRouterAddress,
            earnedAmt
        );

        if (earnedAddress != token0Address) {
            _safeSwap(
                uniRouterAddress,
                earnedAmt.div(2),
                slippageFactor,
                earnedToToken0Path,
                address(this),
                block.timestamp.add(600)
            );
        }

        if (earnedAddress != token1Address) {
            _safeSwap(
                uniRouterAddress,
                earnedAmt.div(2),
                slippageFactor,
                earnedToToken1Path,
                address(this),
                block.timestamp.add(600)
            );
        }

        uint256 token0Amt = IERC20(token0Address).balanceOf(address(this));
        uint256 token1Amt = IERC20(token1Address).balanceOf(address(this));
        if (token0Amt > 0 && token1Amt > 0) {
            IERC20(token0Address).safeIncreaseAllowance(
                uniRouterAddress,
                token0Amt
            );
            IERC20(token1Address).safeIncreaseAllowance(
```

```
                uniRouterAddress,
                token1Amt
            );
            IPancakeRouter02(uniRouterAddress).addLiquidity(
                token0Address,
                token1Address,
                token0Amt,
                token1Amt,
                0,
                0,
                address(this),
                block.timestamp.add(600)
            );
        }

        lastEarnBlock = block.number;

        _farm();
    }

    function buyBack(uint256 _earnedAmt) internal virtual returns (uint256) {
        if (buyBackRate <= 0) {
            return _earnedAmt;
        }

        uint256 buyBackAmt = _earnedAmt.mul(buyBackRate).div(buyBackRateMax);

        if (earnedAddress == UPAddress) {
            IERC20(earnedAddress).safeTransfer(buyBackAddress, buyBackAmt);
        } else {
            IERC20(earnedAddress).safeIncreaseAllowance(
                uniRouterAddress,
                buyBackAmt
            );

            _safeSwap(
                uniRouterAddress,
                buyBackAmt,
                slippageFactor,
                earnedToUpPath,
                buyBackAddress,
                block.timestamp.add(600)
            );
        }
```

```
        return _earnedAmt.sub(buyBackAmt);
    }

    function distributeFees(uint256 _earnedAmt)
        internal
        virtual
        returns (uint256)
    {
        if (_earnedAmt > 0) {
            if (controllerFee > 0) {
                uint256 fee =
                    _earnedAmt.mul(controllerFee).div(controllerFeeMax);
                IERC20(earnedAddress).safeTransfer(rewardsAddress, fee);
                _earnedAmt = _earnedAmt.sub(fee);
            }
        }

        return _earnedAmt;
    }

    function convertDustToEarned() public virtual whenNotPaused {
        require(isAutoComp, "!isAutoComp");
        require(!isCAKEStaking, "isCAKEStaking");

        // Converts dust tokens into earned tokens, which will be reinvested on the
next earn().

        // Converts token0 dust (if any) to earned tokens
        uint256 token0Amt = IERC20(token0Address).balanceOf(address(this));
        if (token0Address != earnedAddress && token0Amt > 0) {
            IERC20(token0Address).safeIncreaseAllowance(
                uniRouterAddress,
                token0Amt
            );

            // Swap all dust tokens to earned tokens
            _safeSwap(
                uniRouterAddress,
                token0Amt,
                slippageFactor,
                token0ToEarnedPath,
                address(this),
                block.timestamp.add(600)
```

```
                );
            }

            // Converts token1 dust (if any) to earned tokens
            uint256 token1Amt = IERC20(token1Address).balanceOf(address(this));
            if (token1Address != earnedAddress && token1Amt > 0) {
                IERC20(token1Address).safeIncreaseAllowance(
                    uniRouterAddress,
                    token1Amt
                );

                // Swap all dust tokens to earned tokens
                _safeSwap(
                    uniRouterAddress,
                    token1Amt,
                    slippageFactor,
                    token1ToEarnedPath,
                    address(this),
                    block.timestamp.add(600)
                );
            }
        }

        function pause() public virtual onlyAllowGov {
            _pause();
        }

        function unpause() public virtual onlyAllowGov {
            _unpause();
        }

        function setSettings(
            uint256 _entranceFeeFactor,
            uint256 _withdrawFeeFactor,
            uint256 _controllerFee,
            uint256 _buyBackRate,
            uint256 _slippageFactor
        ) public virtual onlyAllowGov {
            require(
                _entranceFeeFactor >= entranceFeeFactorLL,
                "_entranceFeeFactor too low"
            );
            require(
                _entranceFeeFactor <= entranceFeeFactorMax,
```

```
            "_entranceFeeFactor too high"
        );
        entranceFeeFactor = _entranceFeeFactor;

        require(
            _withdrawFeeFactor >= withdrawFeeFactorLL,
            "_withdrawFeeFactor too low"
        );
        require(
            _withdrawFeeFactor <= withdrawFeeFactorMax,
            "_withdrawFeeFactor too high"
        );
        withdrawFeeFactor = _withdrawFeeFactor;

        require(_controllerFee <= controllerFeeUL, "_controllerFee too high");
        controllerFee = _controllerFee;

        require(_buyBackRate <= buyBackRateUL, "_buyBackRate too high");
        buyBackRate = _buyBackRate;

        require(
            _slippageFactor <= slippageFactorUL,
            "_slippageFactor too high"
        );
        slippageFactor = _slippageFactor;

        emit SetSettings(
            _entranceFeeFactor,
            _withdrawFeeFactor,
            _controllerFee,
            _buyBackRate,
            _slippageFactor
        );
    }

    function setGov(address _govAddress) public virtual onlyAllowGov {
        govAddress = _govAddress;
        emit SetGov(_govAddress);
    }

    function setOnlyGov(bool _onlyGov) public virtual onlyAllowGov {
        onlyGov = _onlyGov;
        emit SetOnlyGov(_onlyGov);
    }
```

```
    function setUniRouterAddress(address _uniRouterAddress)
        public
        virtual
        onlyAllowGov
    {
        uniRouterAddress = _uniRouterAddress;
        emit SetUniRouterAddress(_uniRouterAddress);
    }

    function setBuyBackAddress(address _buyBackAddress)
        public
        virtual
        onlyAllowGov
    {
        buyBackAddress = _buyBackAddress;
        emit SetBuyBackAddress(_buyBackAddress);
    }

    function setRewardsAddress(address _rewardsAddress)
        public
        virtual
        onlyAllowGov
    {
        rewardsAddress = _rewardsAddress;
        emit SetRewardsAddress(_rewardsAddress);
    }

    function inCaseTokensGetStuck(
        address _token,
        uint256 _amount,
        address _to
    ) public virtual onlyAllowGov {
        require(_token != earnedAddress, "!safe");
        require(_token != wantAddress, "!safe");
        IERC20(_token).safeTransfer(_to, _amount);
    }

    function _wrapBNB() internal virtual {
        // BNB -> WBNB
        uint256 bnbBal = address(this).balance;
        if (bnbBal > 0) {
            IWBNB(wbnbAddress).deposit{value: bnbBal}(); // BNB -> WBNB
        }
```

```
    }

    function wrapBNB() public virtual onlyAllowGov {
        _wrapBNB();
    }

    function _safeSwap(
        address _uniRouterAddress,
        uint256 _amountIn,
        uint256 _slippageFactor,
        address[] memory _path,
        address _to,
        uint256 _deadline
    ) internal virtual {
        uint256[] memory amounts =
            IPancakeRouter02(_uniRouterAddress).getAmountsOut(_amountIn,
_path);
        uint256 amountOut = amounts[amounts.length.sub(1)];

        IPancakeRouter02(_uniRouterAddress)
            .swapExactTokensForTokensSupportingFeeOnTransferTokens(
            _amountIn,
            amountOut.mul(_slippageFactor).div(1000),
            _path,
            _to,
            _deadline
        );
    }
}
```

stakingRewards: fork the marsecosystem’ s pool without the access control.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.6.12;
pragma experimental ABIEncoderV2;

import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Pausable.sol";
```

```
import "./interfaces/ILiquidityMiningMaster.sol";
import "./interfaces/IVestingMaster.sol";

// Earn XMS
contract StakingRewards is
    ILiquidityMiningMaster,
    Ownable,
    ReentrancyGuard,
    Pausable,
    ERC20
{
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    IVestingMaster public vestingMaster;

    // Info of each pool.
    PoolInfo[] public override poolInfo;

    // Info of each user that stakes LP tokens.
    mapping(uint256 => mapping(address => UserInfo)) public override userInfo;

    // Pair corresponding pid
    mapping(address => uint256) public override pair2Pid;
    mapping(IERC20 => bool) public override poolExistence;

    // UP tokens created per block.
    uint256 public override upPerBlock;

    // Bonus muliplier for early UP makers.
    uint256 public constant override BONUS_MULTIPLIER = 1;

    // Total allocation points. Must be the sum of all allocation points in all pools.
    uint256 public override totalAllocPoint = 0;

    // The block number when UP mining starts.
    uint256 public override startBlock;

    // The block number when UP mining ends.
    uint256 public override endBlock;

    address public up;
```

```
    IERC20 public uptoken;

    constructor(
        address _up,
        address _vestingMaster,
        uint256 _upPerBlock,
        uint256 _startBlock,
        uint256 _endBlock
    ) public ERC20("UP Farms Seed Token", "UPSEED") {
        require(
            _startBlock < _endBlock,
            "StakingReward::constructor: End less than start"
        );
        up =_up;
        uptoken = IERC20(up);
        vestingMaster = IVestingMaster(_vestingMaster);
        upPerBlock = _upPerBlock;
        startBlock = _startBlock;
        endBlock = _endBlock;
    }

    modifier nonDuplicated(IERC20 _lpToken) {
        require(
            !poolExistence[_lpToken],
            "StakingReward::nonDuplicated: Duplicated"
        );
        _;
    }

    modifier validatePid(uint256 _pid) {
        require(
            _pid < poolInfo.length,
            "StakingReward::validatePid: Not exist"
        );
        _;
    }

    function poolLength() public view override returns (uint256) {
        return poolInfo.length;
    }

    // Add a new lp to the pool. Can only be called by the governor.
    function addPool(
        uint256 _allocPoint,
```

```
        IERC20 _lpToken,
        bool _withUpdate
    ) public onlyOwner nonDuplicated(_lpToken) {
        require(
            block.number < endBlock,
            "StakingReward::addPool: Exceed endblock"
        );
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock
            ? block.number
            : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolExistence[_lpToken] = true;
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accUPPerShare: 0
            })
        );
        pair2Pid[address(_lpToken)] = poolLength() - 1;
    }

    // Update the given pool's UP allocation point and deposit fee. Can only be called
by the governor.
    function setPool(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public validatePid(_pid) onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    // Return reward multiplier over the given _from to _to block.
    function getMultiplier(uint256 _from, uint256 _to)
```

```
        public
        pure
        override
        returns (uint256)
    {
        return _to.sub(_from).mul(BONUS_MULTIPLIER);
    }

    function getUPReward(uint256 _pid)
        internal
        view
        returns (uint256 upReward)
    {
        PoolInfo storage pool = poolInfo[_pid];
        require(
            pool.lastRewardBlock < block.number,
            "StakingReward::getUPReward: Must little than the current block number"
        );
        uint256 multiplier = getMultiplier(
            pool.lastRewardBlock,
            block.number >= endBlock ? endBlock : block.number
        );
        upReward = multiplier.mul(upPerBlock).mul(pool.allocPoint).div(
            totalAllocPoint
        );
    }

    // View function to see pending UP on frontend.
    function pendingUP(uint256 _pid, address _user)
        external
        view
        override
        validatePid(_pid)
        returns (uint256)
    {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accUPPerShare = pool.accUPPerShare;
        uint256 lpSupply = address(uptoken) == address(pool.lpToken)
            ? totalSupply()
            : pool.lpToken.balanceOf(address(this));
        if (block.number > pool.lastRewardBlock && lpSupply != 0) {
            uint256 shareReward = getUPReward(_pid);
```

```
            accUPPerShare = accUPPerShare.add(
                shareReward.mul(1e12).div(lpSupply)
            );
        }
        return user.amount.mul(accUPPerShare).div(1e12).sub(user.rewardDebt);
    }

    // Update reward variables for all pools. Be careful of gas spending!
    function massUpdatePools() public override {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            updatePool(pid);
        }
    }

    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) public override validatePid(_pid) {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        if (pool.lastRewardBlock >= endBlock) {
            return;
        }
        uint256 lpSupply = address(uptoken) == address(pool.lpToken)
            ? totalSupply()
            : pool.lpToken.balanceOf(address(this));
        uint256 lastRewardBlock = block.number >= endBlock
            ? endBlock
            : block.number;
        if (lpSupply == 0 || pool.allocPoint == 0) {
            pool.lastRewardBlock = lastRewardBlock;
            return;
        }
        uint256 shareReward = getUPReward(_pid);
        pool.accUPPerShare = pool.accUPPerShare.add(
            shareReward.mul(1e12).div(lpSupply)
        );
        pool.lastRewardBlock = lastRewardBlock;
    }

    // Deposit LP tokens to StakingReward for UP allocation.
    function deposit(uint256 _pid, uint256 _amount)
        public
```

```
        override
        validatePid(_pid)
        nonReentrant
        whenNotPaused
    {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user
                .amount
                .mul(pool.accUPPerShare)
                .div(1e12)
                .sub(user.rewardDebt);
            if (pending > 0) {
                uint256 locked;
                if (address(vestingMaster) != address(0)) {
                    locked = pending
                        .div(vestingMaster.lockedPeriodAmount() + 1)
                        .mul(vestingMaster.lockedPeriodAmount());
                }
                safeUPTransfer(msg.sender, pending.sub(locked));
                if (locked > 0) {
                    uint256 actualAmount = safeUPTransfer(
                        address(vestingMaster),
                        locked
                    );
                    vestingMaster.lock(msg.sender, actualAmount);
                }
            }
        }
        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(
                address(msg.sender),
                address(this),
                _amount
            );
            if (address(uptoken) == address(pool.lpToken)) {
                _mint(msg.sender, _amount);
            }
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accUPPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
```

```
    }

    // Withdraw LP tokens from StakingReward.
    function withdraw(uint256 _pid, uint256 _amount)
        public
        override
        validatePid(_pid)
        nonReentrant
    {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(
            user.amount >= _amount,
            "StakingReward::withdraw: Not good"
        );
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user
                .amount
                .mul(pool.accUPPerShare)
                .div(1e12)
                .sub(user.rewardDebt);
            if (pending > 0) {
                uint256 locked;
                if (address(vestingMaster) != address(0)) {
                    locked = pending
                        .div(vestingMaster.lockedPeriodAmount() + 1)
                        .mul(vestingMaster.lockedPeriodAmount());
                }
                safeUPTransfer(msg.sender, pending.sub(locked));
                if (locked > 0) {
                    uint256 actualAmount = safeUPTransfer(
                        address(vestingMaster),
                        locked
                    );
                    vestingMaster.lock(msg.sender, actualAmount);
                }
            }
        }
        if (_amount > 0) {
            user.amount = user.amount.sub(_amount);
            if (address(uptoken) == address(pool.lpToken)) {
                _burn(msg.sender, _amount);
            }
```

```
                pool.lpToken.safeTransfer(address(msg.sender), _amount);
            }
            user.rewardDebt = user.amount.mul(pool.accUPPerShare).div(1e12);
            emit Withdraw(msg.sender, _pid, _amount);
        }

        // Withdraw without caring about rewards. EMERGENCY ONLY.
        function emergencyWithdraw(uint256 _pid)
            public
            override
            validatePid(_pid)
            nonReentrant
        {
            PoolInfo storage pool = poolInfo[_pid];
            UserInfo storage user = userInfo[_pid][msg.sender];
            uint256 amount = user.amount;
            user.amount = 0;
            user.rewardDebt = 0;
            if (address(uptoken) == address(pool.lpToken)) {
                _burn(msg.sender, amount);
            }
            pool.lpToken.safeTransfer(address(msg.sender), amount);
            emit EmergencyWithdraw(msg.sender, _pid, amount);
        }

        // Safe UP transfer function, just in case if rounding error causes pool to not have
enough UP.
        function safeUPTransfer(address _to, uint256 _amount)
            internal
            returns (uint256)
        {
            uint256 balance = uptoken.balanceOf(address(this));
            uint256 amount;
            uint256 floorAmount = totalSupply();
            if (balance > floorAmount) {
                if (_amount > balance.sub(floorAmount)) {
                    amount = balance.sub(floorAmount);
                } else {
                    amount = _amount;
                }
            }
            require(
                uptoken.transfer(_to, amount),
                "StakingReward::safeUPTransfer: Transfer failed"
```

```
        );
        return amount;
    }

    function updateUpPerBlock(uint256 _upPerBlock)
        public
        onlyOwner
    {
        massUpdatePools();
        upPerBlock = _upPerBlock;
        emit UpdateEmissionRate(msg.sender, _upPerBlock);
    }

    function updateEndBlock(uint256 _endBlock)
        public
        onlyOwner
    {
        require(
            _endBlock > startBlock,
            "StakingReward::updateEndBlock: Less"
        );
        for (uint256 pid = 0; pid < poolInfo.length; ++pid) {
            require(
                _endBlock > poolInfo[pid].lastRewardBlock,
                "StakingReward::updateEndBlock: Less"
            );
        }
        massUpdatePools();
        endBlock = _endBlock;
        emit UpdateEndBlock(msg.sender, _endBlock);
    }

    function updateVestingMaster(address _vestingMaster)
        public
        onlyOwner
    {
        vestingMaster = IVestingMaster(_vestingMaster);
        emit UpdateVestingMaster(msg.sender, _vestingMaster);
    }

    function setPause() public onlyOwner {
        _pause();
    }
```

```
    function setUnPause() public onlyOwner {
        _unpause();
    }
}
```

vestingMaster, fork the marsecosystem’ s VestingMaster.sol without Access control.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.12;
pragma experimental ABIEncoderV2;

import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/access/AccessControl.sol";

import "./interfaces/IVestingMaster.sol";

contract VestingMaster is IVestingMaster, ReentrancyGuard, AccessControl {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    struct LockedReward {
        uint256 locked;
        uint256 timestamp;
    }
    // XMS token, or may be other token
    IERC20 public vestingToken;

    mapping(address => LockedReward[]) public userLockedRewards;

    uint256 public immutable period;

    uint256 public immutable override lockedPeriodAmount;

    uint256 public totalLockedRewards;

    bytes32 public constant FARMS_ROLE = keccak256("FARMS_ROLE");

    constructor(
        uint256 _period,
        uint256 _lockedPeriodAmount,
        address _vestingToken
    )public {
```

```
        require(
            _vestingToken != address(0),
            "VestingMaster::constructor: Zero address"
        );
        require(_period > 0, "VestingMaster::constructor: Period zero");
        require(
            _lockedPeriodAmount > 0,
            "VestingMaster::constructor: Period amount zero"
        );
        vestingToken = IERC20(_vestingToken);
        period = _period;
        lockedPeriodAmount = _lockedPeriodAmount;
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
    }

    function lock(address account, uint256 amount) public override{
        require(hasRole(FARMS_ROLE, msg.sender), "Caller is not a farms");

        LockedReward[] memory oldLockedRewards = userLockedRewards[account];
        uint256 currentTimestamp = block.timestamp;
        LockedReward memory lockedReward;
        uint256 claimableAmount;
        for (uint256 i = 0; i < oldLockedRewards.length; i++) {
            lockedReward = oldLockedRewards[i];
            if (currentTimestamp >= lockedReward.timestamp) {
                claimableAmount = claimableAmount.add(lockedReward.locked);
                delete oldLockedRewards[i];
            } else {
                break;
            }
        }

        uint256 newStartTimestamp = (currentTimestamp / period) * period;
        uint256 newTimestamp;
        LockedReward memory newLockedReward;
        uint256 jj = 0;
        delete userLockedRewards[account];
        if (claimableAmount > 0) {
            userLockedRewards[account].push(
                LockedReward({
                    locked: claimableAmount,
                    timestamp: newStartTimestamp
                })
            );
```

```
            }
            for (uint256 i = 0; i < lockedPeriodAmount; i++) {
                newTimestamp = newStartTimestamp.add((i + 1) * period);
                uint256 locked;
                if (amount.div(lockedPeriodAmount) == 0) {
                    locked = i == 0 ? amount : 0;
                } else if (amount.mod(lockedPeriodAmount) > 0) {
                    locked = i == 0
                        ? amount.div(lockedPeriodAmount).add(
                            amount.mod(lockedPeriodAmount)
                        )
                        : amount.div(lockedPeriodAmount);
                } else {
                    locked = amount.div(lockedPeriodAmount);
                }
                newLockedReward = LockedReward({
                    locked: locked,
                    timestamp: newTimestamp
                });
                for (uint256 j = jj; j < oldLockedRewards.length; j++) {
                    lockedReward = oldLockedRewards[j];
                    if (lockedReward.timestamp == newTimestamp) {
                        newLockedReward.locked = newLockedReward.locked.add(
                            lockedReward.locked
                        );
                        jj = j + 1;
                        break;
                    }
                }
                if (newLockedReward.locked > 0) {
                    userLockedRewards[account].push(newLockedReward);
                }
            }
            totalLockedRewards = totalLockedRewards.add(amount);
            emit Lock(account, amount);
        }

        function claim() public override {
            LockedReward[] storage lockedRewards = userLockedRewards[msg.sender];
            uint256 currentTimestamp = block.timestamp;
            LockedReward memory lockedReward;
            uint256 claimableAmount;
            for (uint256 i = 0; i < lockedRewards.length; i++) {
                lockedReward = lockedRewards[i];
```

```
                if (currentTimestamp >= lockedReward.timestamp) {
                    claimableAmount = claimableAmount.add(lockedReward.locked);
                    delete lockedRewards[i];
                } else {
                    break;
                }
            }
            totalLockedRewards = totalLockedRewards.sub(claimableAmount);
            _safeTransfer(msg.sender, claimableAmount);
            emit Claim(msg.sender, claimableAmount);
        }

        function getVestingAmount()
            public
            view
            override
            returns (uint256 lockedAmount, uint256 claimableAmount)
        {
            LockedReward[] memory lockedRewards = userLockedRewards[msg.sender];
            uint256 currentTimestamp = block.timestamp;
            LockedReward memory lockedReward;
            for (uint256 i = 0; i < lockedRewards.length; i++) {
                lockedReward = lockedRewards[i];
                if (currentTimestamp >= lockedReward.timestamp) {
                    claimableAmount = claimableAmount.add(lockedReward.locked);
                } else {
                    lockedAmount = lockedAmount.add(lockedReward.locked);
                }
            }
        }

        function _safeTransfer(address _to, uint256 _amount) internal virtual {
            vestingToken.safeTransfer(_to, _amount);
        }

        event Lock(address account,uint256 amount);
        event Claim(address account,uint256 amount);
}
```