MoonRose Finance

Smart Contract Audit

- MoonRose Finance Audit Summary
- MoonRose Finance Audit
  - Document information
    - Audit results
    - Audited target file
  - Vulnerability analysis
    - Vulnerability distribution
    - Summary of audit results
    - Contract file
    - Analysis of audit results
      - Re-Entrancy
      - Arithmetic Over/Under Flows
      - Unexpected Blockchain Currency
      - Delegatecall
      - Default Visibilities
      - Entropy Illusion
      - External Contract Referencing
      - Unsolved TODO comments
      - Short Address/Parameter Attack
      - Unchecked CALL Return Values
      - Race Conditions / Front Running
      - Denial Of Service (DOS)
      - Block Timestamp Manipulation
      - Constructors with Care
      - Unintialised Storage Pointers
      - Floating Points and Numerical Precision
      - tx.origin Authentication
      - Permission restrictions

MoonRose Finance Audit Summary

Project name : MoonRose Finance Contract

Project address: None

Code URL : https://github.com/MoonRoseFinance/Contract

Commit : e94a24ee088669958c38cc10c58c18f4bd36dc0a

Project target : MoonRose Finance Contract Audit

Blockchain : Moonriver

Test result : PASSED

Audit Info

Audit NO : 0X202111100026

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

MoonRose Finance Audit

The MoonRose Finance team asked us to review and audit their MoonRose Finance contract. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
MoonRose Finance Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-11-10

Audit results

Notice:

```
Pledge mining contract transactions can be restricted by project pledge and withdrawal.
```

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the MoonRose Finance contract. The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and smart contract, this audit report is valid for 3 months from the date of output.

Disclaimer

Armors Labs Reports is not and should not be regarded as an "approval" or "disapproval" of any particular project or team. These reports are not and should not be regarded as indicators of the economy or value of any "product" or "asset" created by any team. Armors do not cover testing or auditing the integration with external contract or services (such as Unicrypt, Uniswap, PancakeSwap etc'…)

Armors Labs Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Armors does not guarantee the safety or functionality of the technology agreed to be analyzed.

Armors Labs postulates that the information provided is not missing, tampered, deleted or hidden. If the information provided is missing, tampered, deleted, hidden or reflected in a way that is not consistent with the actual situation, Armors Labs shall not be responsible for the losses and adverse effects caused. Armors Labs Audits should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

Audited target file

file	md5
ReInvestPool.sol	184a345a0f8b829c1a20c699f8b3fd06
Migrations.sol	4e130d4ae05e6a9df846f54c8c9c9a6a
StakePool.sol	7b47dca78a616b7babbf88b84c0b9123
Third.sol	116f21c8588a41955346b51af2f15bb8
Common.sol	f1a794cee4deab5386b109db50febd54
LinkTokenInterface.sol	7c52ff8c2c26b9c72fc37faaa3b3f24f
ERC677Receiver.sol	611a8f63a75be3712d72ea90f50ef34f
iswap.sol	2e1f4781b3a09abdfd202da0373b6d74
ERC677.sol	134f6d253ae8942b8290e087e2433d9a
Operator.sol	813efb566b62e8acb14b0e405190df4c

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Unexpected Blockchain Currency	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe
Race Conditions / Front Running	safe
Denial Of Service (DOS)	safe
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe
Permission restrictions	safe

Contract file

ReInvestPool.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;
import "./interface/iswap.sol";
import "./Third.sol";
import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/utils/EnumerableSet.sol";
import "@uniswap/v2-core/contracts/interfaces/IUniswapV2Pair.sol";

// MasterChef is the master of RIT. He can make RIT and he is a fair guy.
//
// Note that it's ownable and the owner wields tremendous power. The ownership
// will be transferred to a governance smart contract once RIT is sufficiently
// distributed and the community can show to govern itself.
//
// Have fun reading it. Hopefully it's bug-free. God bless.
contract ReInvestPool is Third {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;
    IUniswapV2Router02 router;
    // Info of each uRIT.
    struct URITInfo {
        uint256 amount;     // How many LP tokens the uRIT has provided.
```

```
        uint256 rewardDebt; // Reward debt. See explanation below.
        uint256 rewardLpDebt; // 已经分的lp利息.
        uint256 lockTime;
        //
        // We do some fancy math here. Basically, any point in time, the amount of RITs
        // entitled to a uRIT but is pending to be distributed is:
        //
        //   pending reward = (uRIT.amount * pool.accRITPerShare) - uRIT.rewardDebt
        //
        // Whenever a uRIT deposits or withdraws LP tokens to a pool. Here's what happens:
        //   1. The pool's `accRITPerShare` (and `lastRewardBlock`) gets updated.
        //   2. URIT receives the pending reward sent to his/her address.
        //   3. URIT's `amount` gets updated.
        //   4. URIT's `rewardDebt` gets updated.
    }

    // Info of each pool.
    struct PoolInfo {
        IERC20 lpToken;           // Address of LP token contract.
        uint256 allocPoint;       // How many allocation points assigned to this pool. RITs to distri
        uint256 lastRewardBlock;  // Last block number that RITs distribution occurs.
        uint256 accRITPerShare; // Accumulated RITs per share, times 1e12. See below.
        uint256 minAMount;
        uint256 maxAMount;
        ERC20 rewardToken;
        uint256 pid;
        uint256 lpSupply;
        uint256 deposit_fee; // 1/10000
        uint256 withdraw_fee; // 1/10000
        uint256 allWithdrawReward;
    }
    uint256 public baseReward = 100000000;
    ISwap public thirdPool;
    // The RIT TOKEN!
    Common public rit;
    // Dev address.
    address public devaddr;
    // Fee address.
    address public feeaddr;
    // RIT tokens created per block.
    uint256 public RITPerBlock;
    // Bonus muliplier for early RIT makers.
    uint256 public constant BONUS_MULTIPLIER = 10;

    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each uRIT that stakes LP tokens.
    mapping (uint256 => mapping (address => URITInfo)) public uRITInfo;
    // Total allocation poitns. Must be the sum of all allocation points in all pools.
    uint256 public totalAllocPoint = 0;
    uint256 public fee = 20; // 30% of profit
    uint256 public feeBase = 100; // 1% of profit

    event Deposit(address indexed uRIT, uint256 indexed pid, uint256 amount);
    event Withdraw(address indexed uRIT, uint256 indexed pid, uint256 amount);
    event ReInvest(uint256 indexed pid);
    event SetDev(address indexed devAddress);
    event SetFee(address indexed feeAddress);
    event SetRITPerBlock(uint256 _RITPerBlock);
    event SetPool(uint256 pid ,address lpaddr,uint256 point,uint256 min,uint256 max);
    constructor(
        Common _rit,
        address _feeaddr,
        address _devaddr,
        uint256 _RITPerBlock,
        IUniswapV2Router02 _router,
```

```
        ISwap _pool
    ) public {
        rit = _rit;
        feeaddr = _feeaddr;
        devaddr = _devaddr;
        RITPerBlock = _RITPerBlock;
        router = _router;
        thirdPool = _pool;
        initRouters();
    }

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

    function setBaseReward(uint256 _base) public onlyOwner {
        baseReward = _base;
    }

    function setRITPerBlock(uint256 _RITPerBlock) public onlyOwner {
        RITPerBlock = _RITPerBlock;
        emit SetRITPerBlock(_RITPerBlock);
    }

    function setFee(uint256 _fee) public onlyOwner {
        require(_fee <=30,"fee can not more than 30%");
        fee = _fee;
    }

    function GetPoolInfo(uint256 id) external view returns (PoolInfo memory) {
        return poolInfo[id];
    }

    function GetURITInfo(uint256 id,address addr) external view returns (URITInfo memory) {
        return uRITInfo[id][addr];
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(uint256 _pid,uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate,uint256 _min,uin
        if (_withUpdate) {
            massUpdatePools();
        }
        require(_deposit_fee <=30,"fee can not more than 3%");
        require(_withdraw_fee <=30,"fee can not more than 3%");
        uint256 lastRewardBlock = block.number;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accRITPerShare: 0,
            minAMount:_min,
            maxAMount:_max,
            rewardToken:_rewardToken,
            pid:_pid,
            lpSupply:0,
            deposit_fee:_deposit_fee,
            withdraw_fee:_withdraw_fee,
            allWithdrawReward:0
        }));
        approve(poolInfo[poolInfo.length-1]);
        emit SetPool(poolInfo.length-1 , address(_lpToken), _allocPoint, _min, _max);
    }

    // Update the given pool's RIT allocation point. Can only be called by the owner.
```

```
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate,uint256 _min,uint256 _max,uint25
        if (_withUpdate) {
            massUpdatePools();
        }
        require(_deposit_fee <=30,"fee can not more than 3%");
        require(_withdraw_fee <=30,"fee can not more than 3%");
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
        poolInfo[_pid].minAMount = _min;
        poolInfo[_pid].maxAMount = _max;
        poolInfo[_pid].deposit_fee = _deposit_fee;
        poolInfo[_pid].withdraw_fee = _withdraw_fee;
        emit SetPool(_pid , address(poolInfo[_pid].lpToken), _allocPoint, _min, _max);
    }

    // Return reward multiplier over the given _from to _to block.
    function getMultiplier(uint256 _from, uint256 _to) public pure returns (uint256) {
        return _to.sub(_from);
    }

    // View function to see pending RITs on frontend.
    function pending(uint256 _pid, address _uRIT) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        URITInfo storage uRIT = uRITInfo[_pid][_uRIT];
        uint256 accRITPerShare = pool.accRITPerShare;
        uint256 lpSupply = pool.lpSupply;
        if (block.number > pool.lastRewardBlock && lpSupply != 0) {
            uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
            uint256 RITReward = multiplier.mul(RITPerBlock).mul(pool.allocPoint).div(totalAllocPoint)
            accRITPerShare = accRITPerShare.add(RITReward.mul(1e12).div(lpSupply));
        }
        return uRIT.amount.mul(accRITPerShare).div(1e12).sub(uRIT.rewardDebt);
    }

       // View function to see pending RITs on frontend.
    function rewardLp(uint256 _pid, address _user) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        URITInfo storage uRIT = uRITInfo[_pid][_user];
        if(thirdPool.userInfo(pool.pid, address(this)).amount <= 0){
            return 0;
        }
        uint256 ba = getWithdrawBalance(_pid, userShares[_pid][_user], thirdPool.userInfo(pool.pid, a
        if(ba > uRIT.amount){
            return ba.sub(uRIT.amount);
        }
        return 0;
    }

    // View function to see pending RITs on frontend.
        // View function to see pending RITs on frontend.
    function allRewardLp(uint256 _pid) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        if(thirdPool.userInfo(pool.pid, address(this)).amount<=pool.lpSupply){
            return 0;
        }
        return pool.allWithdrawReward.add(thirdPool.userInfo(pool.pid, address(this)).amount.sub(pool
    }

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

```
    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid,uint256 _amount,bool isAdd) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        pool.lpSupply = isAdd ? pool.lpSupply.add(_amount) : pool.lpSupply.sub(_amount);
        if (pool.lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 RITReward = multiplier.mul(RITPerBlock).mul(pool.allocPoint).div(totalAllocPoint);

        rit.mint(address(this), RITReward); // Liquidity reward
        pool.accRITPerShare = pool.accRITPerShare.add(RITReward.mul(1e12).div(pool.lpSupply));
        pool.lastRewardBlock = block.number;
    }

    // Deposit LP tokens to MasterChef for RIT allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        require(pause==0,'can not execute');
        PoolInfo storage pool = poolInfo[_pid];
        URITInfo storage uRIT = uRITInfo[_pid][msg.sender];
        updatePool(_pid, 0, true);
        harvest(_pid);// 剩余利息进行复投
        uint256 pendingT = uRIT.amount.mul(pool.accRITPerShare).div(1e12).sub(uRIT.rewardDebt);
        if(pendingT > 0) {
            safeRITTransfer(msg.sender, pendingT);
        }
        if(_amount > 0) { //
            // 先将金额抵押到合约
            if(pool.deposit_fee > 0){
                uint256 feeR = _amount.mul(pool.deposit_fee).div(10000);
                pool.lpToken.safeTransferFrom(address(msg.sender), devaddr, feeR);
                _amount = _amount.sub(feeR);
            }
            uint256 _before = thirdPool.userInfo(pool.pid,address(this)).amount;
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            thirdPool.deposit(pool.pid, _amount);
            uRIT.amount = uRIT.amount.add(_amount);

            if (pool.minAMount > 0 && uRIT.amount < pool.minAMount){
                revert("amount is too low");
            }
            if (pool.maxAMount > 0 && uRIT.amount > pool.maxAMount){
                revert("amount is too high");
            }
            uint256 _after = thirdPool.userInfo(pool.pid,address(this)).amount;
            pool.lpSupply = pool.lpSupply.add(_amount);
            _mint(_pid, _after.sub(_before), msg.sender, _before);
        }
        uRIT.rewardDebt = uRIT.amount.mul(pool.accRITPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // execute when only bug occur
    function safeWithdraw(uint256 _pid) public onlyOwner{
        require(pause==1,'can not execute');
        PoolInfo storage pool = poolInfo[_pid];
        thirdPool.withdraw(pool.pid, pool.lpSupply);
        pool.lpToken.safeTransfer(address(msg.sender), pool.lpSupply);
        uint256 ba = pool.rewardToken.balanceOf(address(this));
        //
        if(ba<=0){
            return;
```

```
        }
        pool.rewardToken.transfer(devaddr,ba);
    }

    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public {
        require(pause==0,'can not execute');
        PoolInfo storage pool = poolInfo[_pid];
        URITInfo storage uRIT = uRITInfo[_pid][msg.sender];
        require(uRIT.amount >= _amount, "withdraw: not good");
        updatePool(_pid, 0, false);

        uint256 pendingT = uRIT.amount.mul(pool.accRITPerShare).div(1e12).sub(uRIT.rewardDebt);
        if(pendingT > 0) {
            safeRITTransfer(msg.sender, pendingT);
        }
        if(_amount > 0) {
            uint256 fene = thirdPool.userInfo(pool.pid,address(this)).amount;
            uint256 _shares = getWithdrawShares(_pid, _amount, msg.sender, uRIT.amount);
            uint256 should_withdraw = getWithdrawBalance(_pid, _shares, fene);
            pool.lpSupply = pool.lpSupply.sub(_amount);
            uRIT.amount = uRIT.amount.sub(_amount);
            thirdPool.withdraw(pool.pid, should_withdraw); //
            if(pool.withdraw_fee>0){
                uint256 needFee = _amount.mul(pool.withdraw_fee).div(10000);
                _amount = _amount.sub(needFee);
                pool.lpToken.safeTransfer(devaddr, needFee);
            }
            //
            safeLpTransfer(_pid,address(msg.sender), _amount);
            _burn(_pid, _shares, msg.sender);
        }
        harvest(_pid);
        //
        uRIT.rewardDebt = uRIT.amount.mul(pool.accRITPerShare).div(1e12);
        emit Withdraw(msg.sender, _pid, _amount);
    }

        // Safe RIT transfer function, just in case if rounding error causes pool to not have enough
    function safeLpTransfer(uint256 _pid,address _to, uint256 _amount) internal {
        PoolInfo storage pool = poolInfo[_pid];
        uint256 RITBal = pool.lpToken.balanceOf(address(this));
        if(RITBal>_amount){
            pool.allWithdrawReward = pool.allWithdrawReward.add(RITBal.sub(_amount));
        }
        pool.lpToken.transfer(_to, RITBal);
    }

    function approve(PoolInfo memory pool) private {
        pool.rewardToken.approve(address(router),uint256(-1));
        address token0 = IUniswapV2Pair(address(pool.lpToken)).token0();
        address token1 = IUniswapV2Pair(address(pool.lpToken)).token1();
        IERC20(token0).approve(address(router),uint256(-1));
        IERC20(token1).approve(address(router),uint256(-1));
        pool.lpToken.approve(address(thirdPool), uint256(-1));
    }

    //
    function calcProfit(uint256 _pid) private{
        PoolInfo storage pool = poolInfo[_pid];
        thirdPool.deposit(pool.pid, 0); //
        uint256 ba = pool.rewardToken.balanceOf(address(this));
        if(ba<baseReward){
            return;
        }
        // pool.rewardToken.transfer(devaddr,ba);
```

```
        uint256 profitFee = ba.mul(fee).div(feeBase);
        pool.rewardToken.transfer(feeaddr,profitFee);
        ba = ba.sub(profitFee);
        uint256 half = ba.div(2);
        ba = ba.sub(half);
        if(half<=0 || ba<=0){
            return;
        }

        address token0 = IUniswapV2Pair(address(pool.lpToken)).token0();
        if(token0 != address(pool.rewardToken)){ //
            swap(router,address(pool.rewardToken),token0,half);
        }

        address token1 = IUniswapV2Pair(address(pool.lpToken)).token1();

        if(token1 != address(pool.rewardToken)){ //
            swap(router,address(pool.rewardToken),token1,ba);
        }

        uint256 token0Ba = IERC20(token0).balanceOf(address(this));
        uint256 token1Ba = IERC20(token1).balanceOf(address(this));
        if( token0Ba <= 0 || token1Ba <= 0 ){ // 没有余额
            return;
        }
        // IERC20(token0).transfer(devaddr,IERC20(token0).balanceOf(address(this)));
        // IERC20(token1).transfer(devaddr,IERC20(token1).balanceOf(address(this)));
        // return;
        (uint256 t0,uint256 t1,) = IUniswapV2Pair(address(pool.lpToken)).getReserves();
        if( t0<=0 || t1<=0 ){ // 没有流动性
            return;
        }
        uint256 out=0;
        uint256 liqui=0;
        if (t0.mul(token1Ba)>token0Ba.mul(t1)){

            out = token0Ba.mul(t1).div(t0);
            if(out <= 0){
                return;
            }

            (,,liqui) = router.addLiquidity(token0, token1, token0Ba, out, 0, 0, address(this), now.a
        } else{ //
            out = token1Ba.mul(t0).div(t1);
            if(out <= 0){
                return;
            }
            (,,liqui) = router.addLiquidity(token0, token1, out, token1Ba, 0, 0, address(this), now.a
        }
        futou(pool); //
    }

    function futou(PoolInfo memory pool) private {
        uint256 ba = pool.lpToken.balanceOf(address(this));
        if(ba<=0){
            return;
        }
        if(pool.lpSupply<=0){
            pool.lpToken.transfer(feeaddr,ba);
            return;
        }
        // pool.lpToken.transfer(devaddr,ba);
        thirdPool.deposit(pool.pid,ba);
    }
```

```
    // auto reinvest
    function harvest(uint256 _pid) public {
        calcProfit(_pid); //
        emit ReInvest(_pid);
    }

    // Safe RIT transfer function, just in case if rounding error causes pool to not have enough RITs
    function safeRITTransfer(address _to, uint256 _amount) internal {
        uint256 RITBal = rit.balanceOf(address(this));
        if(RITBal<=0){
            return;
        }
        if (_amount > RITBal) {
            rit.transfer(_to, RITBal);
        } else {
            rit.transfer(_to, _amount);
        }
    }

    // Update dev address by the previous dev.
    function dev(address _devaddr) public {
        require(msg.sender == devaddr, "dev: wut?");
        require(_devaddr != address(0), "_devaddr is address(0)");
        devaddr = _devaddr;
        emit SetDev(_devaddr);
    }

    // Update fee address by the previous dev.
    function setFeeAddr(address _feeaddr) public {
        require(msg.sender == feeaddr, "fee: wut?");
        require(_feeaddr != address(0), "_feeaddr is address(0)");
        feeaddr = _feeaddr;
        emit SetFee(_feeaddr);
    }
}
```

Migrations.sol

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.22 <0.8.0;

contract Migrations {
    address public owner;
    uint256 public last_completed_migration;

    constructor() public {
        owner = msg.sender;
    }

    modifier restricted() {
        if (msg.sender == owner) _;
    }

    function setCompleted(uint256 completed) public restricted {
        last_completed_migration = completed;
    }
}
```

StakePool.sol

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

```
pragma experimental ABIEncoderV2;

import "./interface/icustom.sol";
import "./Third.sol";
import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/utils/EnumerableSet.sol";
import "@uniswap/v2-core/contracts/interfaces/IUniswapV2Pair.sol";
import "@uniswap/v2-periphery/contracts/interfaces/IUniswapV2Router02.sol";
// MasterChef is the master of REWARD. He can make REWARD and he is a fair guy.
//
// Note that it's ownable and the owner wields tremendous power. The ownership
// will be transferred to a governance smart contract once REWARD is sufficiently
// distributed and the community can show to govern itself.
//
// Have fun reading it. Hopefully it's bug-free. God bless.
contract StakePool is Third {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;
    IUniswapV2Router02 router;
    // Info of each user.
    struct UserInfo {
        uint256 amount;     // How many LP tokens the user has provided.
        uint256 rewardDebt; // Reward debt. See explanation below.
        //
        // We do some fancy math here. Basically, any point in time, the amount of REWARDs
        // entitled to a user but is pending to be distributed is:
        //
        //   pending reward = (user.amount * pool.accREWARDPerShare) - user.rewardDebt
        //
        // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
        //   1. The pool's `accREWARDPerShare` (and `lastRewardBlock`) gets updated.
        //   2. User receives the pending reward sent to his/her address.
        //   3. User's `amount` gets updated.
        //   4. User's `rewardDebt` gets updated.
    }

    // Info of each pool.
    struct PoolInfo {
        IERC20 lpToken;           // Address of LP token contract.
        uint256 allocPoint;       // How many allocation points assigned to this pool. REWARDs to dis
        uint256 lastRewardBlock;  // Last block number that REWARDs distribution occurs.
        uint256 accREWARDPerShare; // Accumulated REWARDs per share, times 1e12. See below.
        uint256 minAMount;
        uint256 maxAMount;
        uint256 deposit_fee; // 1/10000
        uint256 withdraw_fee; // 1/10000
        ICustom lend; // 1/10000
        IERC20 rewardToken; // 1/10000
        uint256 lpSupply;
        uint256 allWithdrawReward;
    }

    // The REWARD TOKEN!
    Common public REWARD;
    // Fee address.
    address public feeaddr;
    // Dev address.
    address public devaddr;
    // Operation address.
    address public operationaddr;
    // Fund address.
    address public fundaddr;
    // REWARD tokens created per block.
    uint256 public REWARDPerBlock;
    // Bonus muliplier for early REWARD makers.
    uint256 public LockMulti = 1;
```

```
    uint256 public LockTime = 30 days;

    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each user that stakes LP tokens.
    mapping (uint256 => mapping (address => UserInfo)) public userInfo;
    // Total allocation poitns. Must be the sum of all allocation points in all pools.
    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 SetDev(address indexed devAddress);
    event SetREWARDPerBlock(uint256 _REWARDPerBlock);
    event SetMigrator(address _migrator);
    event SetOperation(address _operation);
    event SetFund(address _fund);
    event SetInstitution(address _institution);
    event SetFee(address _feeaddr);
    event SetPool(uint256 pid ,address lpaddr,uint256 point,uint256 min,uint256 max);
    constructor(
        Common _REWARD,
        address _feeaddr,
        address _devaddr,
        address _operationaddr,
        address _fundaddr,
        uint256 _REWARDPerBlock,
        uint256 _LockMulti,
        IUniswapV2Router02 _router
    ) public {
        REWARD = _REWARD;
        devaddr = _devaddr;
        feeaddr = _feeaddr;
        REWARDPerBlock = _REWARDPerBlock;
        operationaddr = _operationaddr;
        fundaddr = _fundaddr;
        router = _router;
        LockMulti = _LockMulti;
    }

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

    function setCbay(Common _cbay) public {
        REWARD = _cbay;
    }


    function setREWARDPerBlock(uint256 _REWARDPerBlock) public onlyOwner {
        REWARDPerBlock = _REWARDPerBlock;
        emit SetREWARDPerBlock(_REWARDPerBlock);
    }

    function setLockMulti(uint256 _lockMulti) public onlyOwner {
        LockMulti = _lockMulti;
    }

    function GetPoolInfo(uint256 id) external view returns (PoolInfo memory) {
        return poolInfo[id];
    }

    function GetUserInfo(uint256 id,address addr) external view returns (UserInfo memory) {
        return userInfo[id][addr];
    }
```

```
    function balanceOfUnderlying(PoolInfo memory pool) public view returns (uint256){
        return pool.lend.updatedSupplyOf(address(this));
    }

    // View function to see pending RITs on frontend.
    function rewardLp(uint256 _pid, address _user) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage uRIT = userInfo[_pid][_user];
        uint256 thirdAllBalance = balanceOfUnderlying(pool);
        if(thirdAllBalance <= 0){
            return 0;
        }
        uint256 ba = uRIT.amount.mul(thirdAllBalance).div(pool.lpSupply);
        if(ba > uRIT.amount){
            return ba.sub(uRIT.amount);
        }
        return 0;
    }

    // View function to see pending RITs on frontend.
    function allRewardLp(uint256 _pid) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        uint256 thirdAllBalance = balanceOfUnderlying(pool);
        if(thirdAllBalance <= pool.lpSupply){
            return 0;
        }
        return pool.allWithdrawReward.add(thirdAllBalance.sub(pool.lpSupply));
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate,uint256 _min,uint256 _max,uin
        if (_withUpdate) {
            massUpdatePools();
        }
        require(_deposit_fee <=30,"fee can not more than 3%");
        require(_withdraw_fee <=30,"fee can not more than 3%");
        uint256 lastRewardBlock = block.number;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accREWARDPerShare: 0,
            minAMount:_min,
            maxAMount:_max,
            deposit_fee : _deposit_fee,
            withdraw_fee : _withdraw_fee,
            lend: _lend,
            rewardToken: _rewardToken,
            lpSupply: 0,
            allWithdrawReward: 0
        }));
        approve(poolInfo[poolInfo.length-1]);
        emit SetPool(poolInfo.length-1 , address(_lpToken), _allocPoint, _min, _max);
    }

    function approve(PoolInfo memory pool) private {
        if(address(pool.lend) != address(0) ){
            if(address(pool.rewardToken) != address(0)){
                pool.rewardToken.approve(address(router),uint256(-1));
                pool.rewardToken.approve(address(pool.lend),uint256(-1));
            }
            pool.lpToken.approve(address(pool.lend), uint256(-1));
        }
    }
```

```
    // Update the given pool's REWARD allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate,uint256 _min,uint256 _max,uint25
        if (_withUpdate) {
            massUpdatePools();
        }
        require(_deposit_fee <=30,"fee can not more than 3%");
        require(_withdraw_fee <=30,"fee can not more than 3%");
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
        poolInfo[_pid].minAMount = _min;
        poolInfo[_pid].maxAMount = _max;
        poolInfo[_pid].deposit_fee = _deposit_fee;
        poolInfo[_pid].withdraw_fee = _withdraw_fee;
        poolInfo[_pid].lend = _lend;
        poolInfo[_pid].rewardToken = _rewardToken;
        approve(poolInfo[_pid]);
        emit SetPool(_pid , address(poolInfo[_pid].lpToken), _allocPoint, _min, _max);
    }

    // Return reward multiplier over the given _from to _to block.
    function getMultiplier(uint256 _from, uint256 _to) public pure returns (uint256) {
        return _to.sub(_from);
    }


    function getApy(uint256 _pid) public view returns (uint256) {
        uint256 yearCount = REWARDPerBlock.mul(86400).div(3).mul(365);
        return yearCount.div(getTvl(_pid));
    }


    function getTvl(uint256 _pid) public view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        (uint256 t1,uint256 t2,) = IUniswapV2Pair(address(pool.lpToken)).getReserves();
        address token0 = IUniswapV2Pair(address(pool.lpToken)).token0();
        uint256 allCount = 0;
        if(token0==address(REWARD)){ // 总成本
            allCount = t1.mul(2);
        } else{
            allCount = t2.mul(2);
        }
        uint256 lpSupply = pool.lpSupply;
        uint256 totalSupply = pool.lpToken.totalSupply();
        return allCount.mul(lpSupply).div(totalSupply);
    }

    // View function to see pending REWARDs on frontend.
    function pendingReward(uint256 _pid, address _user) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accREWARDPerShare = pool.accREWARDPerShare;
        uint256 lpSupply = pool.lpSupply;
        if (block.number > pool.lastRewardBlock && lpSupply != 0) {
            uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
            uint256 REWARDReward = multiplier.mul(REWARDPerBlock).mul(pool.allocPoint).div(totalAlloc
            accREWARDPerShare = accREWARDPerShare.add(REWARDReward.mul(1e12).div(lpSupply));
        }
        uint256 pending = user.amount.mul(accREWARDPerShare).div(1e12).sub(user.rewardDebt);
        return pending.div(LockMulti);
    }

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

```
            updatePool(pid,0,true);
        }
    }

    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid,uint256 _amount,bool isAdd) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        pool.lpSupply = isAdd ? pool.lpSupply.add(_amount) : pool.lpSupply.sub(_amount) ;
        if (pool.lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 REWARDReward = multiplier.mul(REWARDPerBlock).mul(pool.allocPoint).div(totalAllocPoin
        uint256 miningReward = REWARDReward;
        uint256 devReward = miningReward.mul(1875);
        REWARD.mint(devaddr, devReward.div(10000)); // 15% Development
        uint256 oprReward = miningReward.mul(375);
        REWARD.mint(operationaddr, oprReward.div(10000)); // 3% Operation
        REWARD.mint(address(this), REWARDReward); // Liquidity reward
        pool.accREWARDPerShare = pool.accREWARDPerShare.add(REWARDReward.mul(1e12).div(pool.lpSupply)
        pool.lastRewardBlock = block.number;
    }

    // Deposit LP tokens to MasterChef for REWARD allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        require(pause==0,'can not execute');
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid,0,true);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accREWARDPerShare).div(1e12).sub(user.rewardDebt);
            if(pending > 0) {
                safeREWARDTransfer(msg.sender, pending);
            }
        }
        if(_amount > 0) {
            if(pool.deposit_fee > 0){
                uint256 feeR = _amount.mul(pool.deposit_fee).div(10000);
                pool.lpToken.safeTransferFrom(address(msg.sender), devaddr, feeR);
                _amount = _amount.sub(feeR);
            }
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = user.amount.add(_amount);
            if (pool.minAMount > 0 && user.amount < pool.minAMount){
                revert("amount is too low");
            }
            if (pool.maxAMount > 0 && user.amount > pool.maxAMount){
                revert("amount is too high");
            }
            if(address(pool.lend) != address(0)){
                depositLend( pool, _amount);
            }
            pool.lpSupply = pool.lpSupply.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accREWARDPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    function depositLend(PoolInfo memory pool,uint256 _amount) private {
        if(_amount<=0){
            return;
        }
```

```
        pool.lend.mint(_amount);
    }

    function withdrawLend(uint256 _pid,uint256 _amount) private returns(uint256){
        PoolInfo storage pool = poolInfo[_pid];
        require(pool.lpSupply>0,"none pool.lpSupply");
        uint256 allAmount = pool.lend.updatedSupplyOf(address(this));
        uint256 shouldAmount = _amount.mul(allAmount).div(pool.lpSupply);
         if(shouldAmount>_amount){
            pool.allWithdrawReward = pool.allWithdrawReward.add(shouldAmount.sub(_amount));
        }
        //
        pool.lend.redeem(shouldAmount);
        return shouldAmount;
    }

    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid,0,false);
        uint256 pending = user.amount.mul(pool.accREWARDPerShare).div(1e12).sub(user.rewardDebt);
        if(pending > 0) {
            safeREWARDTransfer(msg.sender, pending);
        }
        if(_amount > 0) {
            uint256 shouldAmount = _amount;
            if(address(pool.lend) != address(0)){
                shouldAmount = withdrawLend(_pid,_amount);
            }
            user.amount = user.amount.sub(_amount);
            uint256 originAmount = _amount;
            if(pool.withdraw_fee>0){
                uint256 fee = _amount.mul(pool.withdraw_fee).div(10000);
                _amount = _amount.sub(fee);
                pool.lpToken.safeTransfer(devaddr, fee);
            }
            safeLpTransfer(pool,msg.sender,shouldAmount);
            pool.lpSupply = pool.lpSupply.sub(originAmount);
        }
        user.rewardDebt = user.amount.mul(pool.accREWARDPerShare).div(1e12);
        emit Withdraw(msg.sender, _pid, _amount);
    }

    function safeLpTransfer(PoolInfo memory pool,address _to, uint256 _amount) internal {

        uint256 ba = pool.lpToken.balanceOf(address(this));

        if (_amount > ba) {
            pool.lpToken.transfer(_to, ba);
        } else {
            pool.lpToken.transfer(_to, _amount);
        }
    }

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

```
    // Safe REWARD transfer function, just in case if rounding error causes pool to not have enough R
    function safeREWARDTransfer(address _to, uint256 _amount) internal {

        uint256 ba = REWARD.balanceOf(address(this));

        if (_amount > ba) {
            REWARD.transfer(_to, ba);
        } else {
            REWARD.transfer(_to, _amount);
        }
    }

    // Update dev address by the previous dev.
    function dev(address _devaddr) public {
        require(msg.sender == devaddr, "dev: wut?");
        require(_devaddr != address(0), "_devaddr is address(0)");
        devaddr = _devaddr;
        emit SetDev(_devaddr);
    }

    // Update operation address by the previous operation.
    function operation(address _opaddr) public {
        require(msg.sender == operationaddr, "operation: wut?");
        require(_opaddr != address(0), "_opaddr is address(0)");
        operationaddr = _opaddr;
        emit SetOperation(_opaddr);
    }

    // Update fund address by the previous fund.
    function fund(address _fundaddr) public {
        require(msg.sender == fundaddr, "fund: wut?");
        require(_fundaddr != address(0), "_fundaddr is address(0)");
        fundaddr = _fundaddr;
        emit SetFund(_fundaddr);
    }

    // Update fee address by the previous institution.
    function setFee(address _feeaddr) public {
        require(msg.sender == feeaddr, "feeaddr: wut?");
        require(_feeaddr != address(0), "_feeaddr is address(0)");
        feeaddr = _feeaddr;
        emit SetFee(_feeaddr);
    }
}
```


Third.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.12;
import "@uniswap/v2-periphery/contracts/interfaces/IUniswapV2Router02.sol";
import "./Common.sol";

contract Third is Ownable {
    using SafeMath for uint256;
    uint256 public pause = 0;

    mapping (address => address) public routers;
    address public USDT = 0x382bB369d343125BfB2117af9c149795C6C65C50;
    address public KST = 0xab0d1578216A545532882e420A8C61Ea07B00B12;

    function setPause(uint256 _pause) public onlyOwner{
        pause = _pause;
    }
```

```
    // execute when major bug appears
    // Prevent problems with third-party platforms
    // safe User assets
    function executeTransaction(address target, uint value, string memory signature, bytes memory dat
        require(pause==1,'can not execute');
        bytes memory callData;
        if (bytes(signature).length == 0) {
            callData = data;
        } else {
            callData = abi.encodePacked(bytes4(keccak256(bytes(signature))), data);
        }
        // solium-disable-next-line security/no-call-value
        (bool success, ) = target.call{value:value}(callData);
        require(success, "Timelock::executeTransaction: Transaction execution reverted.");
    }

    // pid => allShares
    mapping (uint256 => uint256) public allShares;

    // pid => address => shares
    mapping (uint256 => mapping(address => uint256)) public userShares;

    function _burn(uint256 _pid,uint256 _shares,address _user) internal {
        require(allShares[_pid] >= _shares,'allShares not enough!');
        allShares[_pid] = allShares[_pid].sub(_shares);
        require(userShares[_pid][_user] >= _shares,'user shares not enough!');
        userShares[_pid][_user] = userShares[_pid][_user].sub(_shares);
    }

    // _allBalance is the third platform shares
    // _amount is the deposit new shares
    function _mint(uint256 _pid,uint256 _amount,address _user,uint256 _allBalance) internal {
        uint256 _shares = 0;
        if (allShares[_pid] == 0) {
            _shares = _amount;
        } else {
            _shares = (_amount.mul(allShares[_pid])).div(_allBalance);
        }
        allShares[_pid] = allShares[_pid].add(_shares);
        userShares[_pid][_user] = userShares[_pid][_user].add(_shares);
    }

    function getWithdrawBalance(uint256 _pid,uint256 _shares,uint256 _allBalance) public view returns
        return (_allBalance.mul(_shares)).div(allShares[_pid]);
    }

    function getWithdrawShares(uint256 _pid,uint256 _amount,address _user,uint256 _userBalance) publi
        return _amount.mul(userShares[_pid][_user]).div(_userBalance);
    }

    function userSharesReward(uint256 _pid,address _user,uint256 _allReward) public view returns(uint
        return _allReward.mul(userShares[_pid][_user]).div(allShares[_pid]);
    }

    function addRouter(address a,address b) public {
        routers[a] = b;
    }

    function removeRouter(address a) public {
        routers[a] = address(0);
    }

    function swap(IUniswapV2Router02 router,address token0,address token1,uint256 input) internal {
        if(routers[token1]==address(0)){
            address[] memory path = new address[](2);
            path[0] = token0;
```

```
            path[1] = token1;
            router.swapExactTokensForTokens(input, uint256(0), path, address(this), block.timestamp.a
        } else{
            address[] memory path = new address[](3);
            path[0] = token0;
            path[1] = routers[token1];
            path[2] = token1;
            router.swapExactTokensForTokens(input, uint256(0), path, address(this), block.timestamp.a
        }
    }

    function initRouters() internal {
        routers[KST] = USDT;
    }

}
```

Common.sol

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

import "@openzeppelin/contracts/access/Ownable.sol";
import "./interface/ERC677.sol";

// Token with Governance.
contract Common is ERC677("MoonRose", "DIANA"),Ownable {
    mapping(address =>uint256) public miners;
    /**
     * @dev See {ERC20-_mint}.
     *
     * Requirements:
     *
     * - the caller must have the {MinterRole}.
     */
    function mint(address account, uint256 amount) public returns (bool) {
        require(miners[msg.sender]>0,"not have permission!");
        _mint(account, amount);
        _moveDelegates(address(0), _delegates[account], amount);
        return true;
    }
    function addMiner(address account) public onlyOwner returns (bool) {
        miners[account] = 1;
        return true;
    }
    function removeMiner(address account) public onlyOwner returns (bool) {
        require(miners[account]>0,"account is not miner!");
        miners[account] = 0;
        return true;
    }

    // Copied and modified from YAM code:
    // https://github.com/yam-finance/yam-protocol/blob/master/contracts/token/YAMGovernanceStorage.s
    // https://github.com/yam-finance/yam-protocol/blob/master/contracts/token/YAMGovernance.sol
    // Which is copied and modified from COMPOUND:
    // https://github.com/compound-finance/compound-protocol/blob/master/contracts/Governance/Comp.so

    /// @notice A record of each accounts delegate
    mapping (address => address) internal _delegates;

    /// @notice A checkpoint for marking number of votes from a given block
```

```
    struct Checkpoint {
        uint32 fromBlock;
        uint256 votes;
    }

    /// @notice A record of votes checkpoints for each account, by index
    mapping (address => mapping (uint32 => Checkpoint)) public checkpoints;

    /// @notice The number of checkpoints for each account
    mapping (address => uint32) public numCheckpoints;

    /// @notice The EIP-712 typehash for the contract's domain
    bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,uint256 chainId,add

    /// @notice The EIP-712 typehash for the delegation struct used by the contract
    bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address delegatee,uint256 non

    /// @notice A record of states for signing / validating signatures
    mapping (address => uint) public nonces;

      /// @notice An event thats emitted when an account changes its delegate
    event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed to

    /// @notice An event thats emitted when a delegate account's vote balance changes
    event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);

    /**
     * @notice Delegate votes from `msg.sender` to `delegatee`
     * @param delegator The address to get delegatee for
     */
    function delegates(address delegator)
        external
        view
        returns (address)
    {
        return _delegates[delegator];
    }

   /**
    * @notice Delegate votes from `msg.sender` to `delegatee`
    * @param delegatee The address to delegate votes to
    */
    function delegate(address delegatee) external {
        return _delegate(msg.sender, delegatee);
    }

    /**
     * @notice Delegates votes from signatory to `delegatee`
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(
        address delegatee,
        uint nonce,
        uint expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
        external
    {
        bytes32 domainSeparator = keccak256(
```

```
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name())),
                getChainId(),
                address(this)
            )
        );

        bytes32 structHash = keccak256(
            abi.encode(
                DELEGATION_TYPEHASH,
                delegatee,
                nonce,
                expiry
            )
        );

        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                domainSeparator,
                structHash
            )
        );

        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "::delegateBySig: invalid nonce");
        require(now <= expiry, "::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
    }

    /**
     * @notice Gets the current votes balance for `account`
     * @param account The address to get votes balance
     * @return The number of current votes for `account`
     */
    function getCurrentVotes(address account)
        external
        view
        returns (uint256)
    {
        uint32 nCheckpoints = numCheckpoints[account];
        return nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
    }

    /**
     * @notice Determine the prior number of votes for an account as of a block number
     * @dev Block number must be a finalized block or else this function will revert to prevent misin
     * @param account The address of the account to check
     * @param blockNumber The block number to get the vote balance at
     * @return The number of votes the account had as of the given block
     */
    function getPriorVotes(address account, uint blockNumber)
        external
        view
        returns (uint256)
    {
        require(blockNumber < block.number, "::getPriorVotes: not yet determined");

        uint32 nCheckpoints = numCheckpoints[account];
        if (nCheckpoints == 0) {
            return 0;
        }

        // First check most recent balance
```

```
        if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
            return checkpoints[account][nCheckpoints - 1].votes;
        }

        // Next check implicit zero balance
        if (checkpoints[account][0].fromBlock > blockNumber) {
            return 0;
        }

        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
            uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
            Checkpoint memory cp = checkpoints[account][center];
            if (cp.fromBlock == blockNumber) {
                return cp.votes;
            } else if (cp.fromBlock < blockNumber) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        return checkpoints[account][lower].votes;
    }

    function _delegate(address delegator, address delegatee)
        internal
    {
        address currentDelegate = _delegates[delegator];
        uint256 delegatorBalance = balanceOf(delegator); // balance of underlying  (not scaled);
        _delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }

    function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                // decrease old representative
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
                uint256 srcRepNew = srcRepOld.sub(amount);
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                // increase new representative
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
                uint256 dstRepNew = dstRepOld.add(amount);
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }

    function _writeCheckpoint(
        address delegatee,
        uint32 nCheckpoints,
        uint256 oldVotes,
        uint256 newVotes
    )
        internal
    {
        uint32 blockNumber = safe32(block.number, "::_writeCheckpoint: block number exceeds 32 bits")
```

```
        if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
            checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
        } else {
            checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);
            numCheckpoints[delegatee] = nCheckpoints + 1;
        }

        emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
    }

    function safe32(uint n, string memory errorMessage) internal pure returns (uint32) {
        require(n < 2**32, errorMessage);
        return uint32(n);
    }

    function getChainId() internal pure returns (uint) {
        uint256 chainId;
        assembly { chainId := chainid() }
        return chainId;
    }
}
```

LinkTokenInterface.sol

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

interface LinkTokenInterface {
  function allowance(address owner, address spender) external view returns (uint256 remaining);
  function approve(address spender, uint256 value) external returns (bool success);
  function balanceOf(address owner) external view returns (uint256 balance);
  function decimals() external view returns (uint8 decimalPlaces);
  function decreaseApproval(address spender, uint256 addedValue) external returns (bool success);
  function increaseApproval(address spender, uint256 subtractedValue) external;
  function name() external view returns (string memory tokenName);
  function symbol() external view returns (string memory tokenSymbol);
  function totalSupply() external view returns (uint256 totalTokensIssued);
  function transfer(address to, uint256 value) external returns (bool success);
  function transferAndCall(address to, uint256 value, bytes calldata data) external returns (bool suc
  function transferFrom(address from, address to, uint256 value) external returns (bool success);
}
```

ERC677Receiver.sol

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

/**
 * @dev Implementation for a receiving contract to use.
 */
abstract contract ERC677Receiver {
    function onTokenTransfer(
        address _from,
        uint256 _amount,
        bytes memory _data
    )
    public
    virtual
    returns(bool success);
}
```

iswap.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;
interface ISwap {
     // Info of each user.
    struct SwapUserInfo {
        uint256 amount;     // How many LP tokens the user has provided.
        uint256 rewardDebt; // Reward debt. See explanation below.
    }
    function deposit(uint256 _pid, uint256 _amount) external ;
    function withdraw(uint256 _pid, uint256 _amount) external ;
    function userInfo(uint256 _pid,address _user) external view returns (SwapUserInfo memory);
}
```

ERC677.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.6.12;
import './ERC677Receiver.sol';
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract ERC677 is ERC20 {
    constructor(string memory _name, string memory _symbol) ERC20(_name, _symbol) public {}

    function transferAndCall(
        address _receiver,
        uint256 _amount,
        bytes memory _data
    )
        public
        virtual
        returns(bool success)
    {
        require(super.transfer(_receiver, _amount), "ERC20.transfer(): transfer() returned false"); /

        // Check if _receiver is a contract
        // !! Commented out as to not cause issues between smart contracts and EOAs !!
        if(isContract(_receiver)) {
           (bool result,) = _receiver.call(abi.encodeWithSignature(
                ("onTokenTransfer(address,uint256,bytes)"),
                _receiver,
                _amount,
                _data
                )
            );
            return result;
        }
        return true; // The return value is unclear on how to be used.
    }

    /**
     * @dev Returns true if `account` is a contract.
     *
     * !! Commented out as to not cause issues between smart contracts and users !!
     *
     * Written by OpenZeppelin
     * https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v3.3/contracts/utils/Addre
     *
     */
    function isContract(address _account) internal view returns(bool) {
        // This method relies on extcodesize, which returns 0 for contracts in
```

```
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
        // solhint-disable-next-line no-inline-assembly
        assembly { size := extcodesize(_account) }
        return size > 0;
    }
}
```

Operator.sol

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

pragma solidity ^0.6.0;

import '@openzeppelin/contracts/GSN/Context.sol';
import '@openzeppelin/contracts/access/Ownable.sol';

contract Operator is Context, Ownable {
    address private _operator;

    event OperatorTransferred(
        address indexed previousOperator,
        address indexed newOperator
    );

    constructor() internal {
        _operator = _msgSender();
        emit OperatorTransferred(address(0), _operator);
    }

    function operator() public view returns (address) {
        return _operator;
    }

    modifier onlyOperator() {
        require(
            _operator == msg.sender,
            'operator: caller is not the operator'
        );
        _;
    }

    function isOperator() public view returns (bool) {
        return _msgSender() == _operator;
    }

    function transferOperator(address newOperator_) public onlyOwner {
        _transferOperator(newOperator_);
    }

    function _transferOperator(address newOperator_) internal {
        require(
            newOperator_ != address(0),
            'operator: zero address given for new operator'
        );
        emit OperatorTransferred(_operator, newOperator_);
        _operator = newOperator_;
    }
}
```

Analysis of audit results

Re-Entrancy

- **Description:**
  One of the features of smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle Blockchain Currency, and as such often send Blockchain Currency to various external user addresses. The operation of calling external contracts, or sending Blockchain Currency to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the contract to execute further code (i.e. through a fallback function) , including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Arithmetic Over/Under Flows

- **Description:**
  The Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unexpected Blockchain Currency

- **Description:**
  Typically when Blockchain Currency is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where Blockchain Currency can exist in a contract without having executed any code. Contracts which rely on code execution for every Blockchain Currency sent to the contract can be vulnerable to attacks where Blockchain Currency is forcibly sent to a contract.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Delegatecall

- **Description:**
  The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Default Visibilities

- **Description:**
  Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whBlockchain Currency a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the Solidity Docs. Functions default to public allowing users to call them externally. Incorrect use of visibility specifiers can lead to some devestating vulernabilities in smart contracts as will be discussed in this section.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Entropy Illusion

- **Description:**
  All transactions on the blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no rand() function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many ideas have been proposed to address this (see for example, RandDAO or using a chain of Hashes as described by Vitalik in this post).
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

External Contract Referencing

- **Description:**
  One of the benefits of the global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general

operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Short Address/Parameter Attack

- **Description:**
  This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. I add this attack for completeness and to be aware of how parameters can be manipulated in contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls in solidity. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send()) fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Race Conditions / Front Running

- **Description:**
  The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Denial Of Service (DOS)

- **Description:**
  This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these contracts forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Block Timestamp Manipulation

- **Description:**
  Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Constructors with Care

- **Description:**
  Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unintialised Storage Pointers

- **Description:**
  The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately intialising variables.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Floating Points and Numerical Precision

- **Description:**
  As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

tx.origin Authentication

- **Description:**
  Solidity has a global variable, tx.origin which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

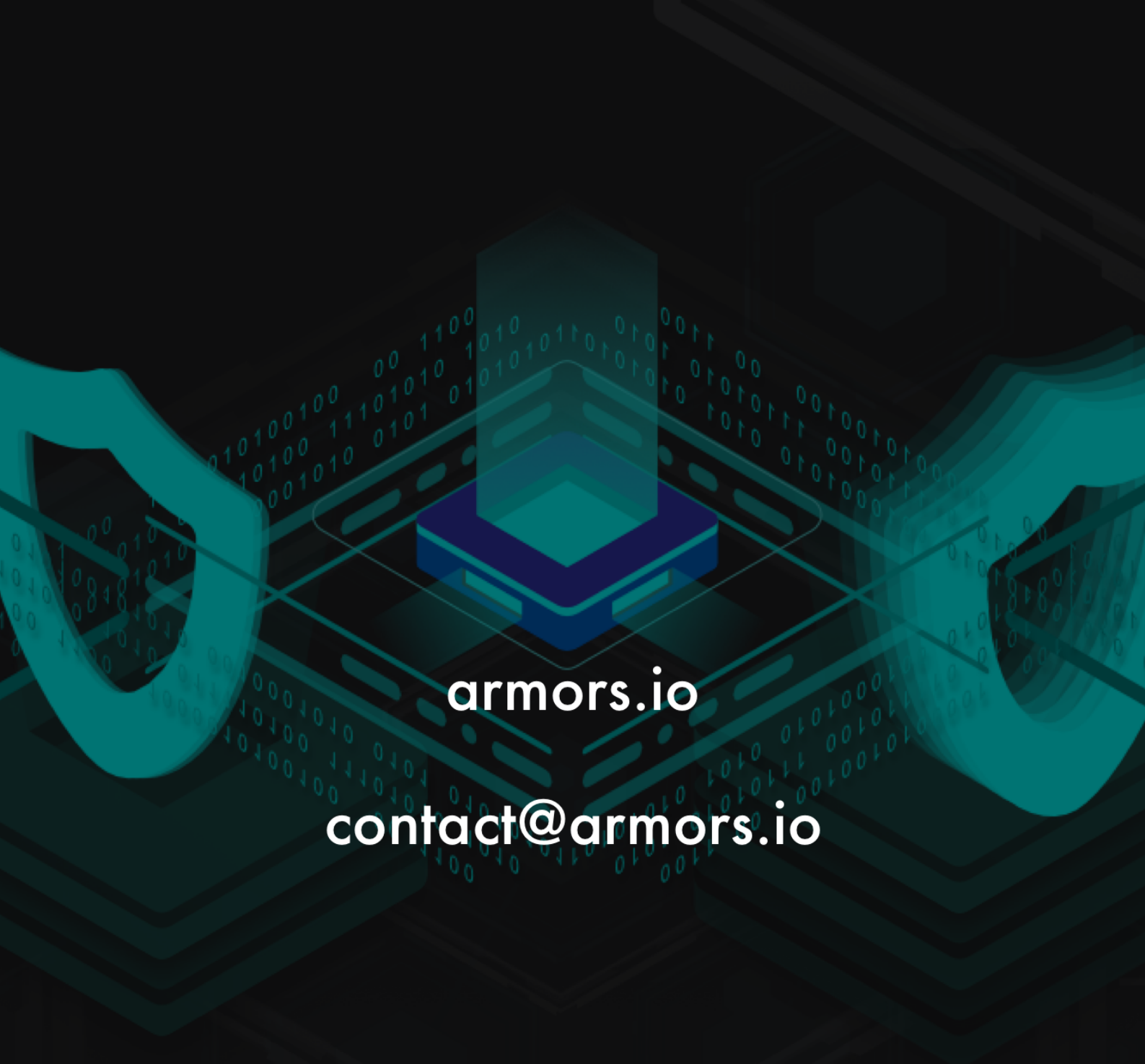
Permission restrictions

- **Description:**
  Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.



armors.io

contact@armors.io