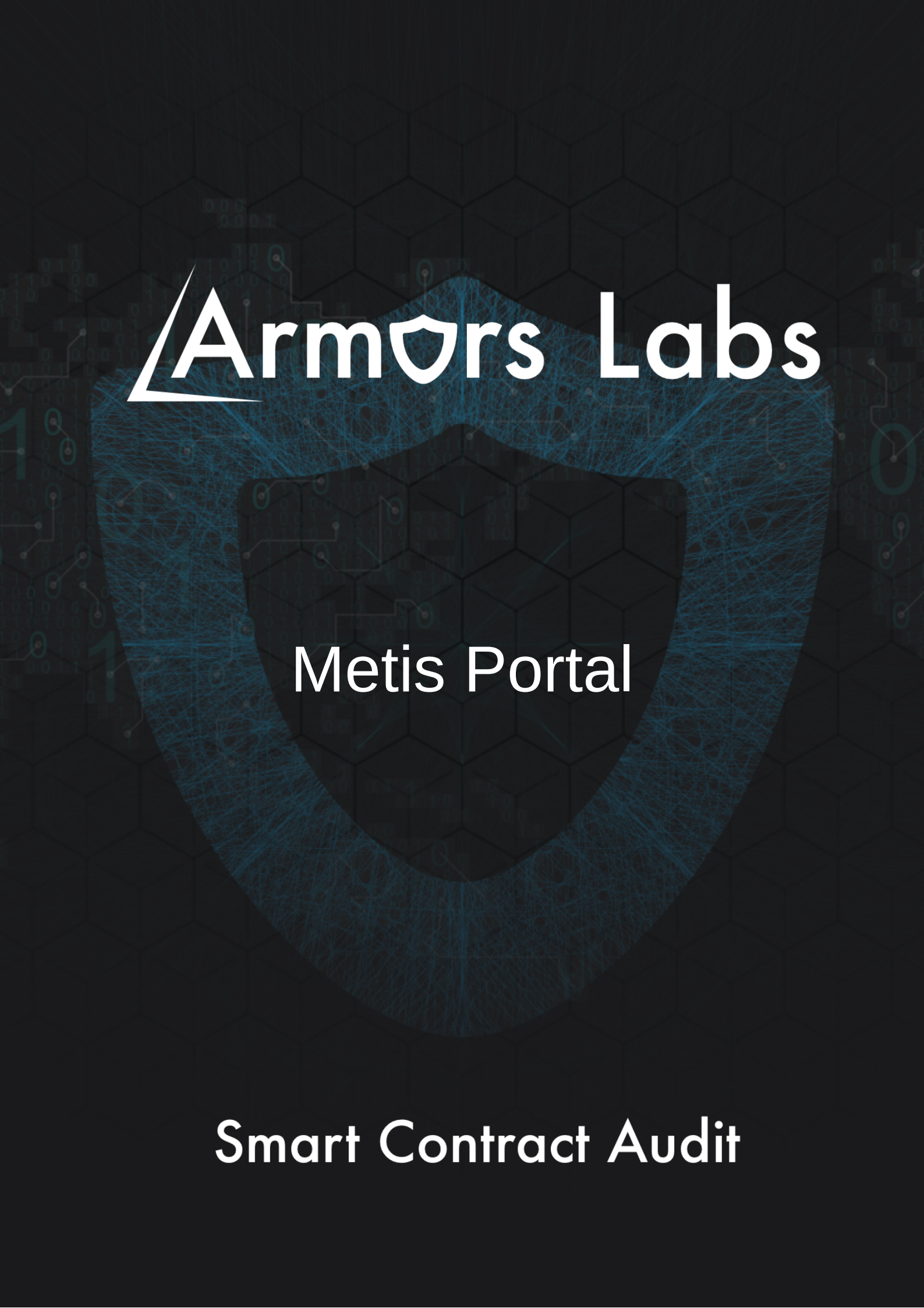
Armors Labs
Metis Portal
Smart Contract Audit

Metis Portal Audit Summary

Project name : Metis Portal Contract

Project address: None

Code URL : https://github.com/MetisProtocol/Metis-Mining

Commit : a943fea02123441b4485650538ab21bc14eb2c70

Project target : Metis Portal Contract Audit

Blockchain : Metis L2

Test result : PASSED

Audit Info

Audit NO : 0X202109180026

Audit Team : Armors Labs

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

Metis Portal Audit

The Metis Portal team asked us to review and audit their Metis Portal 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
Metis Portal Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-09-18

Audit results

Notice：

```
The rescue in the contract Distributor allows the owner to transfer the token to the owner account.
```

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the Metis Portal 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
Vault.sol	25e77a52739900bb5625e24e4d7d6f3c
Distributor.sol	c1396da420285e408c7bf3b0f7690519
Mining.sol	338e8203bd003537917ae3febb9ed669
DACRecorder.sol	66a42d1c68baa9bb901471f2b9199e77
MockMetisToken.sol	0c1b4c638e926bc8dc5528e50f297225
MockDAC.sol	c10b8101494c6270466f774ecadee437
IERC20.sol	dc33f115b06a62f5f6c62af735b8302c
ReentrancyGuard.sol	2f25b3ab814b86c34483953e421d167b
ERC20.sol	dcf8ca8f7bc8229b7d48cf1e42ac1179
Ownable.sol	35a025d046d179f7970ab788cb661bbd
SafeERC20.sol	17f3c1b674724dec96939c1166aa60f9
SafeMath.sol	bb52425389693910db462151d74103f9
Address.sol	fe6002cfe963d27cd7a725ac41f80914
EnumerableSet.sol	c0fd6fc28e102e40601a4b0f771bcab0
Context.sol	e33ce2236c86a1a4273a485b8dd6f9bc
Math.sol	ff55ee5758129908ae3b34db43b31967
IVault.sol	bef9b0ed60fccdb30b21a134160ab5ed
IDAC.sol	2f09a21ad49a6af99447b2acdbc4ca0e
IDACRecorder.sol	f6e96c689bdb01fcaf294feaa5eeab53
IDistributor.sol	d3e2ceb2a250ee6f844e72947460ea91
IMetisToken.sol	a4e137e055b0cf243f8ce64f60b9e93b

file	md5
IMining.sol	ed82100056a3d0aba6202ba23e9de652

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

Vault.sol

```
pragma solidity 0.6.12;

import "./common/IERC20.sol";
import "./common/SafeMath.sol";
import "./common/SafeERC20.sol";

contract Vault {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    address public governance;
    address public DACRecorder;
    IERC20 public metisToken;

    uint public leaveDistributionMin = 1000; // 10% distributed to remaining vault
    uint public constant MAX = 10000;

    // stats tracking
    uint public totalDistributionAmount = 0;

    uint public totalShares;
    mapping(address => uint) public shares;

    /* ========== CONSTRUCTOR ========== */

    constructor(IERC20 _metisToken) public {
        metisToken = _metisToken;
        governance = msg.sender;
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    // Enter staking pool. Pay some Metis. Earn some shares.
    function enter(uint256 _amount, address _user) external onlyRecorder {
        // Gets the amount of Metis locked in the contract
        uint256 totalMetis = metisToken.balanceOf(address(this));
        // If no share exists, mint it 1:1 to the amount put in
        if (totalShares == 0 || totalMetis == 0) {
            mintShares(_user, _amount);
        }
        // Calculate and mint the amount of share the Metis is worth. The ratio will change overtime,
        else {
            uint256 what = _amount.mul(totalShares).div(totalMetis);
            mintShares(_user, what);
        }
        // Lock the Metis in the contract
        metisToken.transferFrom(msg.sender, address(this), _amount);

        emit Enter(_user, _amount);
    }

    // Leave staking pool. Claim back your Metis.
    // Returned Metis = Original Metis + Gained Metis - Penalty
    function leave(uint256 _share) external {
        // Calculates the amount of Metis the share is worth
        uint256 what = _share.mul(metisToken.balanceOf(address(this))).div(totalShares);
        // Calculate penalties
        uint256 distributionAmount = what.mul(leaveDistributionMin).div(MAX);

        burnShares(msg.sender, _share);

        // Transfer to withdrawer
        uint256 withdrawAmount = what.sub(distributionAmount);
```

```
        // update stats: not crutial so do not use SafeMath to avoid overflow revert
        totalDistributionAmount += distributionAmount;

        // send the rest to user
        metisToken.transfer(msg.sender, withdrawAmount);

        emit Leave(msg.sender, what, distributionAmount);
    }

    /* ========== RESTRICTED FUNCTIONS ========== */

    function setGov(address _governance) external onlyGov {
        governance = _governance;
    }

    function setLeaveDistributionMin(uint _leaveDistributionMin) external onlyGov {
        require(_leaveDistributionMin >= 0 && _leaveDistributionMin <= 5000, "leaveDistributionMin is
        leaveDistributionMin = _leaveDistributionMin;
    }

    function setDACRecorder(address _DACRecorder) external onlyGov {
        DACRecorder = _DACRecorder;
    }

    // Allow governance to rescue tokens
    function rescue(address _token) external onlyGov {
        require(_token != address(metisToken), "cannot take user's Metis");
        uint _balance = IERC20(_token).balanceOf(address(this));
        IERC20(_token).safeTransfer(governance, _balance);
    }

    /* ========== MODIFIERS ========== */

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

    modifier onlyRecorder() {
        require(msg.sender == DACRecorder, "!DACRecorder");
        _;
    }

    /* ========== INTERNAL FUNCTIONS ========== */

    function mintShares(address _user, uint _amount) internal {
        totalShares = totalShares.add(_amount);
        shares[_user] = shares[_user].add(_amount);
    }

    function burnShares(address _user, uint _amount) internal {
        shares[_user] = shares[_user].sub(_amount, "burn amount exceeds shares");
        totalShares = totalShares.sub(_amount);
    }

    event Enter(address indexed user, uint256 amount);
    event Leave(address indexed user, uint256 withdrawAmount, uint256 distribution);
}
```


Distributor.sol

```
pragma solidity 0.6.12;
```

```
import "./common/IERC20.sol";
import "./common/SafeERC20.sol";
import "./common/Ownable.sol";

contract Distributor is Ownable {
    using SafeERC20 for IERC20;

    IERC20 public metisToken;
    // Mining Contract
    address public mining;

    constructor(IERC20 _metisToken) public {
        metisToken = _metisToken;
    }

    function distribute(address _to, uint256 _amount) external onlyMining returns (bool) {
        metisToken.safeTransfer(_to, _amount);
        emit Distribute(_to, _amount);
        return true;
    }

    // Allow owner to rescue tokens
    function rescue(address _token) external onlyOwner {
        uint _balance = IERC20(_token).balanceOf(address(this));
        IERC20(_token).safeTransfer(msg.sender, _balance);
    }

    function setMiningContract(address _mining) external onlyOwner {
        mining = _mining;
    }

    /* ========== MODIFIERS ========== */

    modifier onlyMining() {
        require(msg.sender == address(mining), "Not mining contract");
        _;
    }

    event Distribute(address indexed to, uint256 amount);
}
```

Mining.sol

```
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import "./common/IERC20.sol";
import "./common/SafeERC20.sol";
import "./common/SafeMath.sol";
import "./common/Ownable.sol";

import "./interfaces/IMining.sol";
import "./interfaces/IMetisToken.sol";
import "./interfaces/IDAC.sol";
import "./interfaces/IDACRecorder.sol";
import "./interfaces/IDistributor.sol";

contract Mining is Ownable, IMining {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    // Info of each user.
    struct UserInfo {
        uint256 amount; // How many staked tokens the user has provided.
```

```
        uint256 rewardDebt; // Reward debt
    }

    // Info of each pool.
    struct PoolInfo {
        address token; // Address of staked token contract.
        uint256 allocPoint; // How many allocation points assigned to this pool. Metis to distribute
        uint256 lastRewardTimestamp; // Last block timestamp that Metis distribution occurs.
        uint256 accMetisPerShare; // Accumulated Metis per share, times 1e18. See below.
    }

    // The Metis TOKEN!
    IMetisToken public Metis;
    // DAC
    IDAC public DAC;
    // DACRecorder
    IDACRecorder public DACRecorder;
    // Metis distributor
    IDistributor public distributor;
    // Dev address.
    address public teamAddr;
    // Info of each pool.
    PoolInfo[] public poolInfo;
    // mapping of token to pool id
    mapping(address => uint) public override tokenToPid;
    // Info of each user that stakes staked tokens.
    mapping(uint => mapping(address => UserInfo)) public userInfo;
    // Metis tokens created per second.
    uint256 public MetisPerSecond;
    // Total allocation poitns. Must be the sum of all allocation points in all pools.
    uint256 public totalAllocPoint = 0;
    // The block timestamp when Metis mining starts.
    uint256 public startTimestamp;
    uint256 public MIN_DEPOSIT = 10 * 1e18;
    uint256 public MAX_DEPOSIT = 2000 * 1e18;

    /* ========== CONSTRUCTOR ========== */

    constructor(
        IMetisToken _Metis,
        IDACRecorder _DACRecorder,
        IDistributor _distributor,
        uint256 _MetisPerSecond,
        uint256 _startTimestamp
    ) public {
        Metis = _Metis;
        DACRecorder = _DACRecorder;
        distributor = _distributor;
        MetisPerSecond = _MetisPerSecond;
        startTimestamp = _startTimestamp;
    }

    /* ========== VIEW FUNCTIONS ========== */

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

    function calcMetisReward(
        uint256 timestamp,
        uint256 allocPoint
    ) public view returns (uint256 accTime, uint256 MetisReward) {
        accTime = block.timestamp.sub(timestamp);
        MetisReward = MetisPerSecond.mul(accTime).mul(allocPoint).div(totalAllocPoint);
    }
```

```
    // View function to see pending Metis on frontend.
    function pendingMetis(uint256 _dacId, uint256 _pid, address _user) external view returns (uint256
        PoolInfo memory pool = poolInfo[_pid];
        UserInfo memory user = userInfo[_pid][_user];
        (IDACRecorder.DACState dacState,, uint256 accMetisPerShare,,) = DACRecorder.checkDACInfo(_dac
        bool isActiveDAC = dacState == IDACRecorder.DACState.Active;
        uint256 share = isActiveDAC ? pool.accMetisPerShare : accMetisPerShare;
        uint256 totalWeight = DACRecorder.totalWeight();
        if (isActiveDAC && block.timestamp > pool.lastRewardTimestamp && totalWeight != 0) {
            (,uint256 MetisReward) = calcMetisReward(pool.lastRewardTimestamp, pool.allocPoint);
            share = share.add(MetisReward.mul(1e18).div(totalWeight));
        }
        (, uint256 accPower,) = DACRecorder.checkUserInfo(_user);
        return user.amount.mul(accPower).mul(share).div(1e18).sub(user.rewardDebt);
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    // Update reward vairables for all pools. Be careful of gas spending!
    function massUpdatePools() public {
        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 {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.timestamp <= pool.lastRewardTimestamp) {
            return;
        }
        uint256 totalWeight = DACRecorder.totalWeight();
        if (totalWeight == 0) {
            pool.lastRewardTimestamp = block.timestamp;
            return;
        }
        (,uint256 MetisReward) = calcMetisReward(pool.lastRewardTimestamp, pool.allocPoint);
        if (teamAddr != address(0)) {
            distributor.distribute(teamAddr, MetisReward.div(9));
        }
        distributor.distribute(address(DACRecorder), MetisReward);
        emit Mint(MetisReward);
        pool.accMetisPerShare = pool.accMetisPerShare.add(
            MetisReward.mul(1e18).div(totalWeight)
        );
        pool.lastRewardTimestamp = block.timestamp;
    }

    function deposit(
        address _creator,
        address _user,
        uint256 _pid,
        uint256 _amount,
        uint256 _DACMemberCount,
        uint256 _initialDACPower,
        uint256 _dacId
    ) onlyDAC external override returns (bool) {
        (IDACRecorder.DACState dacState,,,,) = DACRecorder.checkDACInfo(_dacId);
        bool isCreator = _creator == address(0);
        address existedCreator = DACRecorder.creatorOf(_user);
        if (isCreator) {
            require(DACRecorder.creatorOf(_user) == address(0), "this user is a member of an existing
            if (!DACRecorder.isCreator(_user)) {
                DACRecorder.addCreator(_user);
            }
```

```
            // else {
            //     require(dacCreator == _user, "this user is not matched to DAC creator");
            // }
        } else {
            // require(DACRecorder.isCreator(_creator), "The creator is not found");
            // require(!DACRecorder.isCreator(_user), "This user is a creator");
            if (existedCreator != address(0)) {
                // old member
                require(_creator == existedCreator, "Wrong creator for this member user");
            } else {
                // new member
                DACRecorder.setCreatorOf(_creator, _user);
                DACRecorder.addMember(_dacId, _user);
            }
        }
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        updatePool(_pid);
        if (user.amount > 0) {
            _sendPending(_dacId, _pid, _user);
        }
        if (_amount > 0 && dacState == IDACRecorder.DACState.Active) {
            uint256 remainingAmount = user.amount.add(_amount);
            require(remainingAmount >= MIN_DEPOSIT && remainingAmount <= MAX_DEPOSIT, "Deposit amount
            user.amount = remainingAmount;
            if (isCreator) {
                DACRecorder.updateCreatorInfo(_user, _dacId, _DACMemberCount, _initialDACPower, user.
            } else {
                DACRecorder.updateMemberInfo(_user, _dacId, _DACMemberCount, _initialDACPower, user.a
            }
            IERC20(pool.token).safeTransferFrom(_user, address(this), _amount);
        }
        (, uint256 accPower,) = DACRecorder.checkUserInfo(_user);
        user.rewardDebt = user.amount.mul(accPower).mul(pool.accMetisPerShare).div(1e18);
        emit Deposit(_creator, _user, _pid, _amount, _DACMemberCount);
        return true;
    }

    function withdraw(
        address _creator,
        uint256 _pid,
        uint256 _amount,
        uint256 _dacId
    ) external override returns (bool) {
        bool isCreator = _creator == address(0);
        if (!isCreator) {
            require(!DACRecorder.isCreator(msg.sender), "This mgs.sender is a creator");
            require(_creator == DACRecorder.creatorOf(msg.sender), "Wrong creator for this member use
        } else {
            require(DACRecorder.isCreator(msg.sender), "The msg.sender is not a creator ");
            require(DACRecorder.creatorOf(msg.sender) == address(0), "this msg.sender is a member of
        }
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        _sendPending(_dacId, _pid, msg.sender);
        if(_amount > 0) {
            uint256 remainingAmount = user.amount.sub(_amount);
            (,uint256 userCount,,,uint256 initialDACPower) = DACRecorder.checkDACInfo(_dacId);
            if (isCreator) {
                if (userCount > DACRecorder.MIN_MEMBER_COUNT() || DACRecorder.DAO_OPEN()) {
                    require(
                        remainingAmount >= MIN_DEPOSIT,
                        "creatorWithdraw: Creator can't dismiss this DAC"
                    );
                }
```

```
                // means that the creator dismiss DAC without DAO opening
                if (remainingAmount == 0) {
                    DACRecorder.updateCreatorInfo(msg.sender, _dacId, 0, 0, remainingAmount, pool.acc
                    DACRecorder.removeCreator(msg.sender);
                    DAC.dismissDAC(msg.sender);
                } else {
                    DACRecorder.updateCreatorInfo(msg.sender, _dacId, userCount, initialDACPower, rem
                }
            } else {
                require(
                    remainingAmount == 0 || remainingAmount >= MIN_DEPOSIT,
                    "Member must left required amount of Metis token or withdraw all"
                );
                // means that the member leave a specific DAC
                if (remainingAmount == 0) {
                    DACRecorder.updateMemberInfo(msg.sender, _dacId, 0, 0, 0, true, false);
                    DACRecorder.subCreatorPower(_dacId, userInfo[_pid][_creator].amount);
                    DACRecorder.delMember(_dacId, msg.sender);
                    DAC.memberLeave(_creator, msg.sender);
                } else {
                    DACRecorder.updateMemberInfo(msg.sender, _dacId, userCount, initialDACPower, rema
                }
            }
            user.amount = remainingAmount;
            IERC20(pool.token).safeTransfer(address(msg.sender), _amount);
        }
        (,uint256 accPower,) = DACRecorder.checkUserInfo(msg.sender);
        user.rewardDebt = user.amount.mul(accPower).mul(pool.accMetisPerShare).div(1e18);
        emit Withdraw(_creator, msg.sender, _pid, _amount);
        return true;
    }

    function dismissDAC(uint256 _dacId, uint256 _pid, address _creator) onlyDAC external override ret
        require(DACRecorder.DAO_OPEN(), "DAO is not opened");
        require(DACRecorder.isCreator(_creator), "dismissDAC: not a creator");
        PoolInfo memory pool = poolInfo[_pid];
        UserInfo storage creator = userInfo[_pid][_creator];
        updatePool(_pid);
        _sendPending(_dacId, _pid, _creator);
        DACRecorder.updateCreatorInfo(_creator, _dacId, 0, 0, creator.amount, pool.accMetisPerShare,
        DACRecorder.removeCreator(_creator);
        DAC.dismissDAC(msg.sender);
        IERC20(pool.token).safeTransfer(_creator, creator.amount);
        creator.amount = 0;
        return true;
    }

    /* ========== INTERNAL FUNCTIONS ========== */

    function _sendPending(uint256 _dacId, uint256 _pid, address _user) internal {
        PoolInfo memory pool = poolInfo[_pid];
        UserInfo memory user = userInfo[_pid][_user];
        (,uint256 accPower,) = DACRecorder.checkUserInfo(_user);
        (IDACRecorder.DACState dacState,, uint256 accMetisPerShare,,) = DACRecorder.checkDACInfo(_dac
        uint256 share = dacState == IDACRecorder.DACState.Active ? pool.accMetisPerShare : accMetisPe
        uint256 pending = user.amount.mul(accPower).mul(share).div(1e18).sub(user.rewardDebt);
        if(pending > 0) {
            DACRecorder.sendRewardToVault(_user, pending);
        }
    }

    /* ========== RESTRICTED FUNCTIONS ========== */

    // Add a new staked token to the pool. Can only be called by the owner.
    function add(uint256 _allocPoint, address _token, bool _withUpdate) external onlyOwner {
        if (_withUpdate) {
```

```
            massUpdatePools();
        }
        uint256 lastRewardTimestamp = block.timestamp > startTimestamp ? block.timestamp : startTimes
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo({
                token: _token,
                allocPoint: _allocPoint,
                lastRewardTimestamp: lastRewardTimestamp,
                accMetisPerShare: 0
            })
        );
        tokenToPid[_token] = poolInfo.length - 1;
    }

    // Update the given pool's Metis allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) external onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    function setMetisPerSecond(uint256 _MetisPerSecond) external onlyOwner {
        massUpdatePools();
        MetisPerSecond = _MetisPerSecond;
    }

    function setMetisToken(IMetisToken _metis) external onlyOwner {
        Metis = _metis;
    }

    function setDAC(IDAC _DAC) external onlyOwner {
        DAC = _DAC;
    }

    function setDACRecorder(IDACRecorder _DACRecorder) external onlyOwner {
        DACRecorder = _DACRecorder;
    }

    function setMinDeposit(uint256 _minDeposit) external onlyOwner {
        MIN_DEPOSIT = _minDeposit;
    }

    function setMaxDeposit(uint256 _maxDeposit) external onlyOwner {
        MAX_DEPOSIT = _maxDeposit;
    }

    function setStartTimestamp(uint256 _startTimestamp) external onlyOwner {
        require(block.number < _startTimestamp, "Cannot change startTimestamp after reward start");
        startTimestamp = _startTimestamp;
        // reinitialize lastRewardTimestamp of all existing pools (if any)
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid <= length; ++pid) {
            PoolInfo storage pool = poolInfo[pid];
            pool.lastRewardTimestamp = _startTimestamp;
        }
    }

    function setTeamAddr(address _teamAddr) external onlyOwner {
        teamAddr = _teamAddr;
    }

    /* ========== MODIFIERS ========== */
```

```
    modifier onlyDAC() {
        require(msg.sender == address(DAC), "only DAC can call this function");
        _;
    }

    /* ========== EVENTS ========== */

    event Deposit(address indexed creator, address indexed user, uint256 pid, uint256 amount, uint256
    event Withdraw(address indexed creator, address indexed user, uint256 pid, uint256 amount);
    event Mint(uint256 amount);
}
```

DACRecorder.sol

```
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import "./common/SafeMath.sol";
import "./common/Ownable.sol";
import "./common/EnumerableSet.sol";
import "./interfaces/IMining.sol";
import "./interfaces/IMetisToken.sol";
import "./interfaces/IVault.sol";
import "./interfaces/IDACRecorder.sol";

contract DACRecorder is Ownable, IDACRecorder {
    using SafeMath for uint256;
    using EnumerableSet for EnumerableSet.AddressSet;

    struct DAC {
        DACState state;
        uint256 id;
        uint256 userCount; // How many users this DAC has
        uint256 accMetisPerShare; // When state is set to inactive, save pool.accMetisPerShare
        uint256 initialDACPower;
        address creator;
        EnumerableSet.AddressSet members;
    }

    struct UserInfo {
        Role userRole;
        uint256 accPower; // Accumulated power for user mining.
        uint256 amount;
    }

    // The Metis TOKEN!
    IMetisToken public Metis;
    // Vault
    IVault public vault;
    // Mining Contract
    IMining public mining;
    // Addresses of creators
    EnumerableSet.AddressSet private creators;
    // DAC id mapping to DAC info
    mapping(uint256 => DAC) private dacInfo;
    // User address mapping to user info
    mapping(address => UserInfo) private userInfo;
    // Return creator of the specific member address
    mapping(address => address) public override creatorOf;
    mapping(address => uint256) public userWeight;

    uint256 public override totalWeight;
    uint256 public MAX_ACC_POWER = 500;
```

```
    uint256 public POWER_STEP_SIZE = 5;
    uint256 public INITIAL_POWER_STEP_SIZE = 10;
    uint256 public MEMBER_POWER = 80;
    uint256 public override MIN_MEMBER_COUNT = 10;
    uint256 public override stakedMetis;
    bool public override DAO_OPEN;

    /* ========== CONSTRUCTOR ========== */

    constructor(IMetisToken _Metis, IVault _vault) public {
        Metis = _Metis;
        vault = _vault;
    }

    /* ========== VIEW FUNCTIONS ========== */

    function checkUserInfo(address _user)
        external
        view
        override
        returns (Role userRole, uint256 accPower, uint256 amount)
    {
        userRole = userInfo[_user].userRole;
        accPower = userInfo[_user].accPower;
        amount = userInfo[_user].amount;
    }

    function checkDACInfo(uint256 _dacId)
        external
        view
        override
        returns (DACState state, uint256 userCount, uint256 accMetisPerShare, address creator, uint25
            state = dacInfo[_dacId].state;
            userCount = dacInfo[_dacId].userCount;
            accMetisPerShare = dacInfo[_dacId].accMetisPerShare;
            creator = dacInfo[_dacId].creator;
            initialDACPower = dacInfo[_dacId].initialDACPower;
    }

    function isCreator(address _user) public view override returns (bool) {
        return creators.contains(_user);
    }

    function getUserAccPower(address _user) external view override returns (uint256) {
        return userInfo[_user].accPower;
    }

    /* ========== INTERNAL FUNCTIONS ========== */

    function _calcAccPowerForCreator(uint256 _initialDACPower, uint256 _DACMemberCount) internal view
        uint256 addedPower = 0;
        if (_DACMemberCount == 2) {
            addedPower = addedPower.add(INITIAL_POWER_STEP_SIZE);
        } else if (_DACMemberCount > 2) {
            uint256 totalStep = _DACMemberCount.sub(2).mul(POWER_STEP_SIZE);
            addedPower = addedPower.add(INITIAL_POWER_STEP_SIZE).add(totalStep);
        }
        accPower = _initialDACPower.add(addedPower);
        accPower = accPower > MAX_ACC_POWER ? MAX_ACC_POWER : accPower;
    }

    function _safeMetisTransferToVault(address _user, uint256 _amount) internal {
        uint256 MetisBal = Metis.balanceOf(address(this));
        if (_amount > MetisBal) {
            Metis.approve(address(vault), MetisBal);
            vault.enter(MetisBal, _user);
```

```
        } else {
            Metis.approve(address(vault), _amount);
            vault.enter(_amount, _user);
        }
    }

    /* ========== RESTRICTED FUNCTIONS ========== */

    function addCreator(address _user) external onlyMining override returns (bool) {
        require(!isCreator(_user), "Duplicate creator");
        creators.add(_user);
        return true;
    }

    function removeCreator(address _user) external onlyMining override returns (bool) {
        require(isCreator(_user), "Creator is not found");
        creators.remove(_user);
        return true;
    }

    function addMember(uint256 _dacId, address _member) external onlyMining override returns (bool) {
        require(dacInfo[_dacId].state == DACState.Active, "DAC is inactive");
        dacInfo[_dacId].members.add(_member);
        return true;
    }

    function delMember(uint256 _dacId, address _member) external onlyMining override returns (bool) {
        dacInfo[_dacId].members.remove(_member);
        return true;
    }

    function updateCreatorInfo(
        address _user,
        uint256 _dacId,
        uint256 _DACMemberCount,
        uint256 _initialDACPower,
        uint256 _amount,
        uint256 _accMetisPerShare,
        bool _withdrawAll
    ) external onlyMining override returns (bool) {
        DAC storage dac = dacInfo[_dacId];
        UserInfo storage user = userInfo[_user];

        require(dac.state == DACState.Active, "This DAC is inactive");

        if (_withdrawAll) {
            // creator withdraw all => dismiss DAC
            totalWeight = totalWeight.sub(userWeight[_user]);
            stakedMetis = stakedMetis.sub(user.amount);
            user.amount = _amount;
            user.userRole = Role.None;
            user.accPower = 0;
            userWeight[_user] = 0;
            dac.userCount = dac.userCount.sub(1);
            dac.state = DACState.Inactive;
            dac.accMetisPerShare = _accMetisPerShare;
        } else {
            // update stakedMetis
            stakedMetis = stakedMetis.sub(user.amount);
            user.amount = _amount;
            stakedMetis = stakedMetis.add(user.amount);
            // update user & dac info
            user.userRole = Role.Creator;
            user.accPower = _calcAccPowerForCreator(_initialDACPower, _DACMemberCount);
            dac.userCount = _DACMemberCount;
            dac.state = DACState.Active;
```

```
            dac.creator = _user;
            dac.initialDACPower = _initialDACPower;
            // update weight info
            totalWeight = totalWeight.sub(userWeight[_user]);
            userWeight[_user] = user.accPower.mul(_amount);
            totalWeight = totalWeight.add(userWeight[_user]);
        }
        return true;
    }

    function updateMemberInfo(
        address _user,
        uint256 _dacId,
        uint256 _DACMemberCount,
        uint256 _initialDACPower,
        uint256 _amount,
        bool _withdrawAll,
        bool _isDeposit
    ) external onlyMining override returns (bool) {
        DAC storage dac = dacInfo[_dacId];
        UserInfo storage user = userInfo[_user];
        UserInfo storage creator = userInfo[dac.creator];

        require(dac.members.contains(_user), "This user is not included in this DAC");

        if (_withdrawAll) {
            totalWeight = totalWeight.sub(userWeight[_user]);
            stakedMetis = stakedMetis.sub(user.amount);
            user.amount = _amount;
            user.userRole = Role.None;
            user.accPower = 0;
            userWeight[_user] = 0;
            creatorOf[_user] = address(0);
            dac.userCount = dac.userCount.sub(1);
        } else {
            if (_isDeposit) {
                require(dac.state == DACState.Active, "This DAC is inactive");
            }
            user.userRole = Role.Member;
            user.accPower = MEMBER_POWER;
            dac.userCount = _DACMemberCount;
            // update stakedMetis
            stakedMetis = stakedMetis.sub(user.amount);
            user.amount = _amount;
            stakedMetis = stakedMetis.add(user.amount);
            userWeight[_user] = user.accPower.mul(_amount);
            totalWeight = totalWeight.add(userWeight[_user]);
            if (creator.accPower > 0) {
                creator.accPower = _calcAccPowerForCreator(_initialDACPower, _DACMemberCount);
                totalWeight = totalWeight.sub(userWeight[dac.creator]);
                userWeight[dac.creator] = creator.accPower.mul(_amount);
                totalWeight = totalWeight.add(userWeight[dac.creator]);
            }
        }
        return true;
    }

    function setCreatorOf(address _creator, address _user) external override onlyMining {
        creatorOf[_user] = _creator;
    }

    function subCreatorPower(uint256 _dacId, uint256 _amount) external onlyMining override returns (b
        DAC storage dac = dacInfo[_dacId];
        if (dac.state == DACState.Inactive) {
            return false;
        }
```

```
        UserInfo storage creator = userInfo[dac.creator];
        if (creator.accPower > 0) {
            uint256 subPower = 0;
            dac.userCount == 1 ? subPower = INITIAL_POWER_STEP_SIZE : subPower = POWER_STEP_SIZE;
            creator.accPower = creator.accPower.sub(subPower);
            totalWeight = totalWeight.sub(userWeight[dac.creator]);
            userWeight[dac.creator] = creator.accPower.mul(_amount);
            totalWeight = totalWeight.add(userWeight[dac.creator]);
        }
        return true;
    }

    function sendRewardToVault(address _user, uint256 _amount) external onlyMining override returns (
        _safeMetisTransferToVault(_user, _amount);
        return true;
    }

    function setMaxAccPower(uint256 _maxAccPower) external onlyOwner {
        MAX_ACC_POWER = _maxAccPower;
    }

    function setPowerStepSize(uint256 _powerStepSize) external onlyOwner {
        POWER_STEP_SIZE = _powerStepSize;
    }

    function setInitialPowerStepSize(uint256 _initialPowerStepSize) external onlyOwner {
        INITIAL_POWER_STEP_SIZE = _initialPowerStepSize;
    }

    function setMemberPower(uint256 _memberPower) external onlyOwner {
        MEMBER_POWER = _memberPower;
    }

    function setMinMemberCount(uint256 _minMemberCount) external onlyOwner {
        MIN_MEMBER_COUNT = _minMemberCount;
    }

    function setDAOOpen(bool _daoOpen) external onlyOwner {
        DAO_OPEN = _daoOpen;
    }

    function setMiningContract(IMining _mining) external onlyOwner {
        mining = _mining;
    }

    function setMetisToken(IMetisToken _metis) external onlyOwner {
        Metis = _metis;
    }

    function setVault(IVault _vault) external onlyOwner {
        vault = _vault;
    }

    /* ========== MODIFIERS ========== */

    modifier onlyMining() {
        require(msg.sender == address(mining), "Not mining contract");
        _;
    }

}
```


MockMetisToken.sol

```
/**
 *Submitted for verification at BscScan.com on 2021-06-11
*/


pragma solidity ^0.6.12;

/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 * Refer from https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/GSN/Contex
 */
contract Context {
    // Empty internal constructor, to prevent people from mistakenly deploying
    // an instance of this contract, which should be used via inheritance.
    constructor () internal { }
    // solhint-disable-previous-line no-empty-blocks

    function _msgSender() internal view returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.co
        return msg.data;
    }
}

pragma solidity ^0.6.12;

/**
 * @dev Interface of the ERC20 standard as defined in the
 * the optional functions; to access them see {ERC20Detailed}.
 */
interface IERC20 {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);

    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
     * Returns a boolean value indicating whether the opera
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, uint256 amount) external returns (bool);
```

```
    /**
     * @dev Returns the remaining number of tokens that `spender` will
     * allowed to spend on behalf of `owner` through {transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are called.
     */
    function allowance(address owner, address spender) external view returns (uint256);

    /**
     * @dev Sets `amount` as the allowance of `spender` over the
     *
     * Returns a boolean value indicating whether the opera
     *
     * IMPORTANT: Beware that changing an allowance with this method brings
     * that someone may use both the old and the new allov
     * transaction ordering. One possible solution to mitigate this race
     * condition is to first reduce the spender's allowance to 0 and set the
     * desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * Emits an {Approval} event.
     */
    function approve(address spender, uint256 amount) external returns (bool);

    /**
     * @dev Moves `amount` tokens from `sender` to `recipient` using the
     * allowance mechanism. `amount` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the opera
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);
}

pragma solidity ^0.6.12;
```

```
/**
 * @dev Optional functions from the ERC20 standard.
 * Refer from https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20D
 */
abstract contract ERC20Detailed is IERC20 {
    string private _name;
    string private _symbol;
    uint8 private _decimals;

    /**
   * @dev Sets the values for `name`, `symbol`, and `decimals`. All three of
   * these values are immutable: they can only be set onc
   * construction.
   */
    constructor (string memory name, string memory symbol, uint8 decimals) public {
        _name = name;
        _symbol = symbol;
        _decimals = decimals;
    }

    /**
   * @dev Returns the name of the token.
   */
    function name() public view returns (string memory) {
        return _name;
    }

    /**
   * @dev Returns the symbol of the token, usually
   * name.
   */
    function symbol() public view returns (string memory) {
        return _symbol;
    }

    /**
   * @dev Returns the number of decimals used to get its user representation.
   * For example, if `decimals` equals `2`, a balance of `505` tokens shou
   * be displayed to a user as `5,05` (`505 / 10 ** 2`).
   *
   * Tokens usually opt for a value of 18, imitating the relat
   * Ether and Wei.
   *
   * NOTE: This information is only used for _display_ purposes: it in
   * no way affects any of the arithmetic of the contract, inc
   * {IERC20-balanceOf} and {IERC20-transfer}.
   */
    function decimals() public view returns (uint8) {
        return _decimals;
    }
}


pragma solidity ^0.6.12;
```

```
/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom mes
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     * - Subtraction cannot overflow.
     *
     * _Available since v2.4.0._
```

```
 */
  function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
      require(b <= a, errorMessage);
      uint256 c = a - b;

      return c;
  }

              /**
 * @dev Returns the multiplication of two unsigned integers, reverting on
 * overflow.
 *
 * Counterpart to Solidity's `*` operator.
 *
 * Requirements:
 * - Multiplication cannot overflow.
 */
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
      // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
      // benefit is lost if 'b' is also tested.
      // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
      if (a == 0) {
          return 0;
      }

      uint256 c = a * b;
      require(c / a == b, "SafeMath: multiplication overflow");

      return c;
  }

              /**
 * @dev Returns the integer division of two unsigned integers. Reverts on
 * division by zero. The result is rounded towards zero.
 *
 * Counterpart to Solidity's `/` operator. Note: this function uses a
 * `revert` opcode (which leaves remaining gas untouched) while Solidity
 * uses an invalid opcode to revert (consuming all remaining gas).
 *
 * Requirements:
 * - The divisor cannot be zero.
 */
  function div(uint256 a, uint256 b) internal pure returns (uint256) {
      return div(a, b, "SafeMath: division by zero");
  }

              /**
 * @dev Returns the integer division of two unsigned integers. Reverts with custom r
 * division by zero. The result is rounded towards zero.
 *
 * Counterpart to Solidity's `/` operator. Note: this function uses a
 * `revert` opcode (which leaves remaining gas untouched) while Solidity
 * uses an invalid opcode to revert (consuming all remaining gas).
 *
 * Requirements:
 * - The divisor cannot be zero.
```

```
     *
     * _Available since v2.4.0._
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b != 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer
     * Reverts with custom message when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     *
     * _Available since v2.4.0._
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}

pragma solidity ^0.6.12;

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclus
 * specific functions.
 *
 * This module is used through inheritance. It will make available the
 * `onlyOwner`, which can be applied to your functions to restrict their use to
```

```
 * the owner.
 */
contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
   * @dev Initializes the contract setting the deployer as
   */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
   * @dev Returns the address of the current owner.
   */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
   * @dev Throws if called by any account other than the owner.
   */
    modifier onlyOwner() {
        require(isOwner(), "Ownable: caller is not the owner");
        _;
    }

    /**
   * @dev Returns true if the caller is the current owner.
   */
    function isOwner() public view returns (bool) {
        return _msgSender() == _owner;
    }

    /**
   * @dev Leaves the contract without owner. It will not b
   * `onlyOwner` functions anymore. Can only be called by the current owner.
   *
   * NOTE: Renouncing ownership will leave
   * thereby removing any functionality that is only available to the owner.
   */
    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
   * @dev Transfers ownership of the contract to a new ac
   * Can only be called by the current owner.
   */
    function transferOwnership(address newOwner) public onlyOwner {
        _transferOwnership(newOwner);
    }
```

```
    /**
     * @dev Transfers ownership of the contract to a new ac
     */
    function _transferOwnership(address newOwner) internal {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

pragma solidity ^0.6.12;

/**
 * @title Roles
 * @dev Library for managing addresses assigned to a Role.
 */
library Roles {
    struct Role {
        mapping (address => bool) bearer;
    }

    /**
     * @dev Give an account access to this role.
     */
    function add(Role storage role, address account) internal {
        require(!has(role, account), "Roles: account already has role");
        role.bearer[account] = true;
    }

    /**
     * @dev Remove an account's access to this role.
     */
    function remove(Role storage role, address account) internal {
        require(has(role, account), "Roles: account does not have role");
        role.bearer[account] = false;
    }

    /**
     * @dev Check if an account has this role.
     * @return bool
     */
    function has(Role storage role, address account) internal view returns (bool) {
        require(account != address(0), "Roles: account is the zero address");
        return role.bearer[account];
    }
}



pragma solidity ^0.6.12;



/**
 * @dev Implementation of the {IERC20} interface.
 *
```

```
 * This implementation is agnostic to the way tokens are cr
 * that a supply mechanism has to be added in a derived co
 * For a generic mechanism see {ERC20Mintable}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * We have followed general OpenZeppelin guidelines: functions revert instead
 * of returning `false` on failure. This behavior is nonetheless conventional
 * and does not conflict with the expectations of ERC20 applications.
 *
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts j
 * by listening to said events. Other implementations of the EIP may not emit
 * these events, as it isn't required by the specification.
 *
 * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
 * functions have been added to mitigate the well-known issues around setting
 * allowances. See {IERC20-approve}.
 * Refer from https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.
 */
contract ERC20 is Context, IERC20 {
    using SafeMath for uint256;

    mapping (address => uint256) private _balances;

    mapping (address => mapping (address => uint256)) private _allowances;

    uint256 private _totalSupply;

                /**
   * @dev See {IERC20-totalSupply}.
   */
    function totalSupply() public override view returns (uint256) {
        return _totalSupply;
    }

                /**
   * @dev See {IERC20-balanceOf}.
   */
    function balanceOf(address account) public override view returns (uint256) {
        return _balances[account];
    }

                /**
   * @dev See {IERC20-transfer}.
   *
   * Requirements:
   *
   * - `recipient` cannot be the zero address.
   * - the caller must have a balance of at least `amount`.
   */
    function transfer(address recipient, uint256 amount) public override returns (bool) {
        _transfer(_msgSender(), recipient, amount);
```

```
        return true;
    }

    /**
     * @dev See {IERC20-allowance}.
     */
    function allowance(address owner, address spender) public override view returns (uint256) {
        return _allowances[owner][spender];
    }

    /**
     * @dev See {IERC20-approve}.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public override returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;
    }

    /**
     * @dev See {IERC20-transferFrom}.
     *
     * Emits an {Approval} event indicating the updated allow
     * required by the EIP. See the note at the
     *
     * Requirements:
     * - `sender` and `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     * - the caller must have allowance for `sender`'s tokens of at least
     * `amount`.
     */
    function transferFrom(address sender, address recipient, uint256 amount) public override returns
        _transfer(sender, recipient, amount);
        _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer
        return true;
    }

    /**
     * @dev Atomically increases the allowance granted to `spender` by t
     *
     * This is an alternative to {approve} that can be used as a
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allow
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
        return true;
```

```
    }

    /**
     * @dev Atomically decreases the allowance granted to `spender` by t
     *
     * This is an alternative to {approve} that can be used as a
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allow
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, "ERC2
        return true;
    }

    /**
     * @dev Moves tokens `amount` from `sender` to `recipient`.
     *
     * This is internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `sender` cannot be the zero address.
     * - `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     */
    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }

    /** @dev Creates `amount` tokens and assigns them to `account`, increasing
     * the total supply.
     *
     * Emits a {Transfer} event with `from` set to the zero add
     *
     * Requirements
     *
     * - `to` cannot be the zero address.
     */
    function _mint(address account, uint256 amount) internal {
```

```
        require(account != address(0), "ERC20: mint to the zero address");

        _totalSupply = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
        emit Transfer(address(0), account, amount);
    }

    /**
     * @dev Destroys `amount` tokens from `account`, reducing the
     * total supply.
     *
     * Emits a {Transfer} event with `to` set to the zero addres
     *
     * Requirements
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
    function _burn(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: burn from the zero address");

        _balances[account] = _balances[account].sub(amount, "ERC20: burn amount exceeds balance");
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(account, address(0), amount);
    }

    /**
     * @dev Sets `amount` as the allowance of `spender` over the
     *
     * This is internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(address owner, address spender, uint256 amount) internal {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    /**
     * @dev Destroys `amount` tokens from `account`.`amount` is then deducted
     * from the caller's allowance.
     *
     * See {_burn} and {_approve}.
     */
    function _burnFrom(address account, uint256 amount) internal {
        _burn(account, amount);
        _approve(account, _msgSender(), _allowances[account][_msgSender()].sub(amount, "ERC20: burn a
```

```
    }
}

pragma solidity ^0.6.12;

contract MockMetisToken is Context, ERC20, ERC20Detailed, Ownable {

    using Roles for Roles.Role;

    Roles.Role private _minters;
    using SafeMath for uint256;

    address[] public minters_;
    uint256 maxSupply_;

    constructor(
         address[] memory minters,
        uint256 maxSupply
    )
       ERC20Detailed("Mock Metis Token", "Metis", 18)
       public
    {
        for (uint256 i = 0; i < minters.length; ++i) {
            _minters.add(minters[i]);
        }
        minters_ = minters;
        maxSupply_ = maxSupply;
    }

    function mint(address target, uint256 amount) external {
        require(_minters.has(msg.sender), "ONLY_MINTER_ALLOWED_TO_DO_THIS");
        require(SafeMath.add(totalSupply(), amount) <= maxSupply_, "EXCEEDING_MAX_SUPPLY");
        _mint(target, amount);
    }

    function burn(address target, uint256 amount) external {
        require(_minters.has(msg.sender), "ONLY_MINTER_ALLOWED_TO_DO_THIS");
        _burn(target, amount);
    }

    function addMinter(address minter) external onlyOwner {
        require(!_minters.has(minter), "HAVE_MINTER_ROLE_ALREADY");
        _minters.add(minter);
        minters_.push(minter);
    }


    function removeMinter(address minter) external onlyOwner {
        require(_minters.has(msg.sender), "HAVE_MINTER_ROLE_ALREADY");
        _minters.remove(minter);
        uint256 i;
        for (i = 0; i < minters_.length; ++i) {
            if (minters_[i] == minter) {
                minters_[i] = address(0);
                break;
            }
        }
    }
}
```

MockDAC.sol

```
pragma solidity 0.6.12;

import "../interfaces/IDAC.sol";
import "../interfaces/IMining.sol";
import "../interfaces/IMetisToken.sol";
import "../common/Ownable.sol";

contract MockDAC is IDAC, Ownable {
    event MemberLeave(address indexed creator, address indexed member);
    event DismissDAC(address indexed creator);

    IMining public miningContract;
    IMetisToken public metis;
    bool public DAO_OPEN = false;

    constructor(IMining _miningContract, IMetisToken _metis) public {
        miningContract = _miningContract;
        metis = _metis;
    }

    function memberLeave(address _creator, address _member) external override returns (bool) {
        emit MemberLeave(_creator, _member);
        return true;
    }

    function creatorDeposit(uint256 _amount, uint256 _DACMemberCount, uint256 _initialDACPower, uint2
        // check allowance of spender
        require(
            metis.allowance(msg.sender, address(miningContract)) >= _amount,
            "Not enough allowance for mining contract"
        );
        miningContract.deposit(
            address(0),
            msg.sender,
            miningContract.tokenToPid(address(metis)),
            _amount,
            _DACMemberCount,
            _initialDACPower,
            _dacId
        );
    }

    function memberDeposit(address _creator, uint256 _amount, uint256 _DACMemberCount, uint256 _initi
        // check allowance of spender
        require(
            metis.allowance(msg.sender, address(miningContract)) >= _amount,
            "Not enough allowance for mining contract"
        );
        miningContract.deposit(
            _creator,
            msg.sender,
            miningContract.tokenToPid(address(metis)),
            _amount,
            _DACMemberCount,
            _initialDACPower,
            _dacId
        );
    }

    function dismissDAC(address _creator) public override returns (bool) {
        emit DismissDAC(_creator);
    }

    function DAODismiss(uint256 _dacId) public returns (bool) {
        require(DAO_OPEN, "DAO is not opened");
```

```
        miningContract.dismissDAC(_dacId, miningContract.tokenToPid(address(metis)), msg.sender);
        emit DismissDAC(msg.sender);
    }

    function setDAOOpen(bool _daoOpen) external onlyOwner {
        DAO_OPEN = _daoOpen;
    }

    function setMiningContract(IMining _miningContract) external onlyOwner {
        miningContract = _miningContract;
    }
}
```

IERC20.sol

```
pragma solidity ^0.6.0;

/**
 * @dev Interface of the ERC20 standard as defined in the
 */
interface IERC20 {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);

    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
     * Returns a boolean value indicating whether the opera
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, uint256 amount) external returns (bool);

    /**
     * @dev Returns the remaining number of tokens that `spender` will
     * allowed to spend on behalf of `owner` through {transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are called.
     */
    function allowance(address owner, address spender) external view returns (uint256);

    /**
     * @dev Sets `amount` as the allowance of `spender` over the
     *
     * Returns a boolean value indicating whether the opera
     *
```

```
    * IMPORTANT: Beware that changing an allowance with this method brings
    * that someone may use both the old and the new allow
    * transaction ordering. One possible solution to mitigate this race
    * condition is to first reduce the spender's allowance to 0 and set the
    * desired value afterwards:
    * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    *
    * Emits an {Approval} event.
    */
    function approve(address spender, uint256 amount) external returns (bool);

    /**
    * @dev Moves `amount` tokens from `sender` to `recipient` using the
    * allowance mechanism. `amount` is then deducted from the caller's
    * allowance.
    *
    * Returns a boolean value indicating whether the opera
    *
    * Emits a {Transfer} event.
    */
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    /**
    * @dev Emitted when `value` tokens are moved from one account (`from`) to
    * another (`to`).
    *
    * Note that `value` may be zero.
    */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
    * @dev Emitted when the allowance of a `spender` for
    * a call to {approve}. `value` is the new allowance.
    */
    event Approval(address indexed owner, address indexed spender, uint256 value);
}
```

ReentrancyGuard.sol

```
pragma solidity ^0.6.0;

/**
 * @dev Contract module that helps prevent reentrant calls to a function.
 *
 * Inheriting from `ReentrancyGuard` will make the `nonR
 * available, which can be aplied to functions to make sure there are no nested
 * (reentrant) calls to them.
 *
 * Note that because there is a single `nonReentrant` guard, functions marked as
 * `nonReentrant` may not call one another. This can be worked around by making
 * those functions `private`, and then adding `external` `nonReentrant` entry
```

```
 * points to them.
 */
contract ReentrancyGuard {
    ///              @dev counter to allow mutex lock with only one SSTORE operation
    uint256 private _guardCounter;

    constructor () internal {
        // The counter starts at one to prevent changing it from zero to a non-zero
        // value, which is a more expensive operation.
        _guardCounter = 1;
    }

                /**
   * @dev Prevents          a          contract from calling itself, directly or indirectly.
   * Calling          a          `nonReentrant` function from another `nonReentrant`
   * function is not supported. It is possible to prevent this from happening
   * by making          the          `nonReentrant` function external, and make it call          a
   * `private` function that does          the          actual work.
   */
    modifier nonReentrant() {
        _guardCounter += 1;
        uint256 localCounter = _guardCounter;
        _;
        require(localCounter == _guardCounter, "ReentrancyGuard: reentrant call");
    }
}
```

ERC20.sol

```
pragma solidity ^0.6.0;

import "./Context.sol";
import "./IERC20.sol";
import "./SafeMath.sol";
import "./Address.sol";

            /**
 * @dev Implementation of          the          {IERC20} interface.
 *
 * This implementation is agnostic to          the          way tokens          are          c
 * that          a          supply mechanism has to be added in          a          derived c
 * For          a          generic mechanism see {ERC20PresetMinterPauser}.
 *
 * TIP: For          a          detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * We have followed general OpenZeppelin guidelines: functions revert instead
 * of returning `false` on failure. This behavior is nonetheless conventional
 * and does not conflict with          the          expectations of ERC20 applications.
 *
 * Additionally,          an          {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct          the          allowance for all accounts
 * by listening to said events. Other implementations of          the          EIP may not emit
```

```
 * these events, as it isn't required by the specification.
 *
 * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
 * functions have been added to mitigate the well-known issues around setting
 * allowances. See {IERC20-approve}.
 */
contract ERC20 is Context, IERC20 {
    using SafeMath for uint256;
    using Address for address;

    mapping (address => uint256) private _balances;

    mapping (address => mapping (address => uint256)) private _allowances;

    uint256 private _totalSupply;

    string private _name;
    string private _symbol;
    uint8 private _decimals;

    /**
     * @dev Sets the values for {name} and {symbol}, initializes {decimals} with
     * a default value of 18.
     *
     * To select a different value for {decimals}, use {_setupDecimals}.
     *
     * All three of these values are immutable: they can only
     * construction.
     */
    constructor (string memory name, string memory symbol) public {
        _name = name;
        _symbol = symbol;
        _decimals = 18;
    }

    /**
     * @dev Returns the name of the token.
     */
    function name() public view returns (string memory) {
        return _name;
    }

    /**
     * @dev Returns the symbol of the token, usually
     * name.
     */
    function symbol() public view returns (string memory) {
        return _symbol;
    }

    /**
     * @dev Returns the number of decimals used to get its user representation.
     * For example, if `decimals` equals `2`, a balance of `505` tokens shou
     * be displayed to a user as `5,05` (`505 / 10 ** 2`).
     *
     * Tokens usually opt for a value of 18, imitating the rela
     * Ether and Wei. This is the value {ERC20} uses, unless {_setupDecimals} is
```

```
 * called.
 *
 * NOTE: This information is only used for _display_ purposes: it in
 * no way affects any of the arithmetic of the contract, inc
 * {IERC20-balanceOf} and {IERC20-transfer}.
 */
  function decimals() public view returns (uint8) {
      return _decimals;
  }

                /**
 * @dev See {IERC20-totalSupply}.
 */
  function totalSupply() public view override returns (uint256) {
      return _totalSupply;
  }

                /**
 * @dev See {IERC20-balanceOf}.
 */
  function balanceOf(address account) public view override returns (uint256) {
      return _balances[account];
  }

                /**
 * @dev See {IERC20-transfer}.
 *
 * Requirements:
 *
 * - `recipient` cannot be the zero address.
 * - the caller must have a balance of at least `amount`.
 */
  function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
      _transfer(_msgSender(), recipient, amount);
      return true;
  }

                /**
 * @dev See {IERC20-allowance}.
 */
  function allowance(address owner, address spender) public view virtual override returns (uint256)
      return _allowances[owner][spender];
  }

                /**
 * @dev See {IERC20-approve}.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
  function approve(address spender, uint256 amount) public virtual override returns (bool) {
      _approve(_msgSender(), spender, amount);
      return true;
  }
```

```
    /**
 * @dev See {IERC20-transferFrom}.
 *
 * Emits an {Approval} event indicating the updated allow
 * required by the EIP. See the note at the
 *
 * Requirements:
 * - `sender` and `recipient` cannot be the zero address.
 * - `sender` must have a balance of at least `amount`.
 * - the caller must have allowance for ``sender``'s tokens of at least
 * `amount`.
 */
  function transferFrom(address sender, address recipient, uint256 amount) public virtual override
      _transfer(sender, recipient, amount);
      _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer
      return true;
  }

    /**
 * @dev Atomically increases the allowance granted to `spender` by t
 *
 * This is an alternative to {approve} that can be used as a
 * problems described in {IERC20-approve}.
 *
 * Emits an {Approval} event indicating the updated allow
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
  function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
      _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
      return true;
  }

    /**
 * @dev Atomically decreases the allowance granted to `spender` by t
 *
 * This is an alternative to {approve} that can be used as a
 * problems described in {IERC20-approve}.
 *
 * Emits an {Approval} event indicating the updated allow
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 * - `spender` must have allowance for the caller of at least
 * `subtractedValue`.
 */
  function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool
      _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, "ERC2
      return true;
  }
```

```
    /**
 * @dev Moves tokens `amount` from `sender` to `recipient`.
 *
 * This is internal function is equivalent to {transfer}, and can be used to
 * e.g. implement automatic token fees, slashing mechanisms, etc.
 *
 * Emits a {Transfer} event.
 *
 * Requirements:
 *
 * - `sender` cannot be the zero address.
 * - `recipient` cannot be the zero address.
 * - `sender` must have a balance of at least `amount`.
 */
  function _transfer(address sender, address recipient, uint256 amount) internal virtual {
      require(sender != address(0), "ERC20: transfer from the zero address");
      require(recipient != address(0), "ERC20: transfer to the zero address");

      _beforeTokenTransfer(sender, recipient, amount);

      _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");
      _balances[recipient] = _balances[recipient].add(amount);
      emit Transfer(sender, recipient, amount);
  }

    /** @dev Creates `amount` tokens and assigns them to `account`, increasing
 *  the total supply.
 *
 * Emits a {Transfer} event with `from` set to the zero adc
 *
 * Requirements
 *
 * - `to` cannot be the zero address.
 */
  function _mint(address account, uint256 amount) internal virtual {
      require(account != address(0), "ERC20: mint to the zero address");

      _beforeTokenTransfer(address(0), account, amount);

      _totalSupply = _totalSupply.add(amount);
      _balances[account] = _balances[account].add(amount);
      emit Transfer(address(0), account, amount);
  }

    /**
 * @dev Destroys `amount` tokens from `account`, reducing the
 * total supply.
 *
 * Emits a {Transfer} event with `to` set to the zero addre
 *
 * Requirements
 *
 * - `account` cannot be the zero address.
 * - `account` must have at least `amount` tokens.
 */
```

```
    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");

        _beforeTokenTransfer(account, address(0), amount);

        _balances[account] = _balances[account].sub(amount, "ERC20: burn amount exceeds balance");
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(account, address(0), amount);
    }

    /**
     * @dev Sets `amount` as the allowance of `spender` over the
     *
     * This is internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(address owner, address spender, uint256 amount) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    /**
     * @dev Sets {decimals} to a value other than the defaul
     *
     * WARNING: This function should only be called from the
     * applications that interact with token contracts will not expect
     * {decimals} to ever change, and may work incorrectly if it does.
     */
    function _setupDecimals(uint8 decimals_) internal {
        _decimals = decimals_;
    }

    /**
     * @dev Hook that is called before any transfer of tokens. This includes
     * minting and burning.
     *
     * Calling conditions:
     *
     * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     *   will be to transferred to `to`.
     * - when `from` is zero, `amount` tokens will be minted for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens will be burned.
     * - `from` and `to` are never both zero.
     *
     * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-ho
```

```
   */
    function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual { }
}
```

Ownable.sol

```
pragma solidity ^0.6.0;

import "./Context.sol";

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }
```

```
    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}
```

SafeERC20.sol

```
pragma solidity ^0.6.0;

import "./IERC20.sol";
import "./SafeMath.sol";
import "./Address.sol";

/**
 * @title SafeERC20
 * @dev Wrappers around ERC20 operations that throw on failure (when the token
 * contract returns false). Tokens that return no value (and instead revert or
 * throw on failure) are also supported, non-reverting calls are assumed to be
 * successful.
 * To use this library you can add a `using SafeERC20 for IERC20;` statement to your contract,
 * which allows you to call the safe operations as `token.safeTransfer(...)`, etc.
 */
library SafeERC20 {
    using SafeMath for uint256;
    using Address for address;

    function safeTransfer(IERC20 token, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));
    }

    function safeTransferFrom(IERC20 token, address from, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to, valu
    }

    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(IERC20 token, address spender, uint256 value) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        // solhint-disable-next-line max-line-length
        require((value == 0) || (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
    }

    function safeIncreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).add(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowan
    }

    function safeDecreaseAllowance(IERC20 token, address spender, uint256 value) internal {
```

```
        uint256 newAllowance = token.allowance(address(this), spender).sub(value, "SafeERC20: decreas
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowan
    }

    /**
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxin
     * on the return value: the return value is optional (but if data is returned, it must not be fal
     * @param token The token targeted by the call.
     * @param data The call data (encoded using abi.encode or one of its variants).
     */
    function _callOptionalReturn(IERC20 token, bytes memory data) private {
        // We need to perform a low level call here, to bypass Solidity's return data size checking m
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which
        // the target address contains contract code and also asserts for success in the low-level ca

        bytes memory returndata = address(token).functionCall(data, "SafeERC20: low-level call failed
        if (returndata.length > 0) { // Return data is optional
            // solhint-disable-next-line max-line-length
            require(abi.decode(returndata, (bool)), "SafeERC20: ERC20 operation did not succeed");
        }
    }
}
```

SafeMath.sol

```
pragma solidity ^0.6.0;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     *
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
```

```
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom mes
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     *
     * - Multiplication cannot overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }
```

```
    /**
     * @dev Returns the integer division of two unsigned integers. Reverts on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom r
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
```

```
    * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer
    * Reverts with custom message when dividing by zero.
    *
    * Counterpart to Solidity's `%` operator. This function uses a `revert`
    * opcode (which leaves remaining gas untouched) while Solidity uses an
    * invalid opcode to revert (consuming all remaining gas).
    *
    * Requirements:
    *
    * - The divisor cannot be zero.
    */
     function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
         require(b != 0, errorMessage);
         return a % b;
     }
}
```

Address.sol

```
pragma solidity ^0.6.2;

            /**
 * @dev Collection of functions related to the address type
 */
library Address {
                /**
    * @dev Returns true if `account` is a contract.
    *
    * [IMPORTANT]
    * ====
    * It is unsafe to assume that an address for which this function returns
    * false is an externally-owned account (EOA) and not a
    *
    * Among others, `isContract` will return false for the fol
    * types of addresses:
    *
    * - an externally-owned account
    * - a contract in construction
    * - an address where a contract will
    * - an address where a contract lived, but
    * ====
    */
     function isContract(address account) internal view returns (bool) {
         // According to EIP-1052, 0x0 is the value returned for not-yet created accounts
         // and 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470 is returned
         // for accounts without code, i.e. `keccak256('')`
         bytes32 codehash;
         bytes32 accountHash = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
         // solhint-disable-next-line no-inline-assembly
         assembly { codehash := extcodehash(account) }
         return (codehash != accountHash && codehash != 0x0);
     }
```

```
    /**
 * @dev Replacement for Solidity's `transfer`: sends `amount` wei to
 * `recipient`, forwarding all available gas and reverting on errors.
 *
 * https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases the gas cost
 * of certain opcodes, possibly making contracts go over the 2300 gas limit
 * imposed by `transfer`, making them unable to receive funds via
 * `transfer`. {sendValue} removes this limitation.
 *
 * https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-now/[Learn more
 *
 * IMPORTANT: because control is transferred to `recipient`, care must be
 * taken to not create reentrancy vulnerabilities. Consider using
 * {ReentrancyGuard} or the
 * https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html#use-the -che
 */
  function sendValue(address payable recipient, uint256 amount) internal {
      require(address(this).balance >= amount, "Address: insufficient balance");

      // solhint-disable-next-line avoid-low-level-calls, avoid-call-value
      (bool success, ) = recipient.call{ value: amount }("");
      require(success, "Address: unable to send value, recipient may have reverted");
  }

    /**
 * @dev Performs a Solidity function call using a low lev
 * plain`call` is an unsafe replacement for a function call:
 * function instead.
 *
 * If `target` reverts with a revert reason, it is bubbled up by this
 * function ( like regular Solidity function calls).
 *
 * Returns the raw returned data. To convert to the expe
 * use https://solidity.readthedocs.io/en/latest/units-and-global-variables.html?highlight=abi.decode#abi-encoding
 *
 * Requirements:
 *
 * - `target` must be a contract.
 * - calling `target` with `data` must not revert.
 *
 * _Available since v3.1._
 */
  function functionCall(address target, bytes memory data) internal returns (bytes memory) {
    return functionCall(target, data, "Address: low-level call failed");
  }

    /**
 * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`], but with
 * `errorMessage` as a fallback revert reason when `target` reverts.
 *
 * _Available since v3.1._
 */
  function functionCall(address target, bytes memory data, string memory errorMessage) internal ret
```

```
        return _functionCallWithValue(target, data, 0, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but also transferring `value` wei to `target`.
     *
     * Requirements:
     *
     * - the calling contract must have an ETH balance of at
     * - the called Solidity function must be `payable`.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(address target, bytes memory data, uint256 value) internal returns
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed"
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValue`],
     * with `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(address target, bytes memory data, uint256 value, string memory er
        require(address(this).balance >= value, "Address: insufficient balance for call");
        return _functionCallWithValue(target, data, value, errorMessage);
    }

    function _functionCallWithValue(address target, bytes memory data, uint256 weiValue, string memor
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: weiValue }(data);
        if (success) {
            return returndata;
        } else {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via assembly

                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }
}
```

EnumerableSet.sol

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

pragma solidity 0.6.12;
```

```
/**
* @dev Library for managing
* https://en.wikipedia.org/wiki/Set_(abstract_data_type)[sets] of primitive
* types.
*
* Sets have the following properties:
*
* - Elements are added, removed, and checked for existence in constant time
* (O(1)).
* - Elements are enumerated in O(n). No guarantees are r
*
*
```

- contract Example {
- // Add the library methods
- using EnumerableSet for EnumerableSet.AddressSet; *
- // Declare a set state variable
- EnumerableSet.AddressSet private mySet;
- }
- ``` *
- As of v3.3.0, sets of type `bytes32` (`Bytes32Set`), `address` (`AddressSet`)
- and `uint256` (`UintSet`) are supported. */ library EnumerableSet { // To implement this library for multiple types with as little code // repetition as possible, we write it in terms of a generic Set type with // bytes32 values. // The Set implementation uses private functions, and user-facing // implementations (such as AddressSet) are just wrappers around the // underlying Set. // This means that we can only create new EnumerableSets for types that fit // in bytes32.

  struct Set {

  ```
  // Storage of set values
  bytes32[] _values;

  // Position of the value in the `values` array, plus 1 because index 0
  // means a value is not in the set.
  mapping (bytes32 => uint256) _indexes;
  ```

  }

  /**

  - @dev Add a value to a set. O(1). *
  - Returns true if the value was added to the set, that is if it was not
  - already present. */ function _add(Set storage set, bytes32 value) private returns (bool) { if (!_contains(set, value)) {

```
set._values.push(value);
// The value is stored at length-1, but we add 1 to all indexes
// and use 0 as a sentinel value
set._indexes[value] = set._values.length;
return true;
```

} else {

```
return false;
```

} }

/**

- @dev Removes a value from a set. O(1). *
- Returns true if the value was removed from the set, that is if it was
- present. */ function _remove(Set storage set, bytes32 value) private returns (bool) { // We read and store the value's index to prevent multiple reads from the same storage slot uint256 valueIndex = set._indexes[value];

  if (valueIndex != 0) { // Equivalent to contains(set, value)

```
// To delete an element from the _values array in O(1), we swap the element to delete with t
// the array, and then remove the last element (sometimes called as 'swap and pop').
// This modifies the order of the array, as noted in {at}.

uint256 toDeleteIndex = valueIndex - 1;
uint256 lastIndex = set._values.length - 1;

// When the value to delete is the last one, the swap operation is unnecessary. However, sin
// so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' statement.

bytes32 lastvalue = set._values[lastIndex];

// Move the last value to the index where the value to delete is
set._values[toDeleteIndex] = lastvalue;
// Update the index for the moved value
set._indexes[lastvalue] = toDeleteIndex + 1; // All indexes are 1-based

// Delete the slot where the moved value was stored
set._values.pop();

// Delete the index for the deleted slot
delete set._indexes[value];

return true;
```

} else {

```
return false;
```

} }

/**

- @dev Returns true if the value is in the set. O(1). */ function _contains(Set storage set, bytes32 value) private view returns (bool) { return set._indexes[value] != 0; }

  /**

- @dev Returns the number of values on the set. O(1). */ function _length(Set storage set) private view returns (uint256) { return set._values.length; }

/**

- @dev Returns the value stored at position `index` in the set. O(1). *
- Note that there are no guarantees on the ordering of values inside the
- array, and it may change when more values are added or removed. *
- Requirements: *
- - `index` must be strictly less than {length}. */ function _at(Set storage set, uint256 index) private view returns (bytes32) { require(set._values.length > index, "EnumerableSet: index out of bounds"); return set._values[index]; }

  // Bytes32Set

  struct Bytes32Set { Set _inner; }

  /**

  - @dev Add a value to a set. O(1). *
  - Returns true if the value was added to the set, that is if it was not
  - already present. */ function add(Bytes32Set storage set, bytes32 value) internal returns (bool) { return _add(set._inner, value); }

  /**

  - @dev Removes a value from a set. O(1). *
  - Returns true if the value was removed from the set, that is if it was
  - present. */ function remove(Bytes32Set storage set, bytes32 value) internal returns (bool) { return _remove(set._inner, value); }

  /**

  - @dev Returns true if the value is in the set. O(1). */ function contains(Bytes32Set storage set, bytes32 value) internal view returns (bool) { return _contains(set._inner, value); }

  /**

  - @dev Returns the number of values in the set. O(1). */ function length(Bytes32Set storage set) internal view returns (uint256) { return _length(set._inner); }

/**

- @dev Returns the value stored at position `index` in the set. O(1). *
- Note that there are no guarantees on the ordering of values inside the
- array, and it may change when more values are added or removed. *

- Requirements: *
- - `index` must be strictly less than {length}. */ function at(Bytes32Set storage set, uint256 index) internal view returns (bytes32) { return _at(set._inner, index); }

  // AddressSet

  struct AddressSet { Set _inner; }

  /**

  - @dev Add a value to a set. O(1). *
  - Returns true if the value was added to the set, that is if it was not
  - already present. */ function add(AddressSet storage set, address value) internal returns (bool) { return _add(set._inner, bytes32(uint256(uint160(value)))); }

  /**

  - @dev Removes a value from a set. O(1). *
  - Returns true if the value was removed from the set, that is if it was
  - present. */ function remove(AddressSet storage set, address value) internal returns (bool) { return _remove(set._inner, bytes32(uint256(uint160(value)))); }

  /**

  - @dev Returns true if the value is in the set. O(1). */ function contains(AddressSet storage set, address value) internal view returns (bool) { return _contains(set._inner, bytes32(uint256(uint160(value)))); }

  /**

  - @dev Returns the number of values in the set. O(1). */ function length(AddressSet storage set) internal view returns (uint256) { return _length(set._inner); }

/**

- @dev Returns the value stored at position `index` in the set. O(1). *
- Note that there are no guarantees on the ordering of values inside the
- array, and it may change when more values are added or removed. *
- Requirements: *
- - `index` must be strictly less than {length}. */ function at(AddressSet storage set, uint256 index) internal view returns (address) { return address(uint160(uint256(_at(set._inner, index)))); }

```
// UintSet

struct UintSet {
    Set _inner;
}

/**
 * @dev Add a value to a set. O(1).
 *
 * Returns true if the value was added to the set, that is if it was not
 * already present.
 */
function add(UintSet storage set, uint256 value) internal returns (bool) {
    return _add(set._inner, bytes32(value));
}
```

```
    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }
```

/**

```
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
```

}

Context.sol

```javascript
pragma solidity ^0.6.0;

/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.co
        return msg.data;
```

```
    }
}
```

Math.sol

```
pragma solidity ^0.6.0;

/**
 * @dev Standard math utilities missing in the Solidity language.
 */
library Math {
    /**
     * @dev Returns the largest of two numbers.
     */
    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a >= b ? a : b;
    }

    /**
     * @dev Returns the smallest of two numbers.
     */
    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a < b ? a : b;
    }

    /**
     * @dev Returns the average of two numbers. The result is rounded towards
     * zero.
     */
    function average(uint256 a, uint256 b) internal pure returns (uint256) {
        // (a + b) / 2 can overflow, so we distribute
        return (a / 2) + (b / 2) + ((a % 2 + b % 2) / 2);
    }
}
```

IVault.sol

```
pragma solidity 0.6.12;

interface IVault {
    function enter(uint256 _amount, address _user) external;
    function leave(uint256 _share) external;
    function shares(address) external view returns (uint);
}
```

IDAC.sol

```
pragma solidity 0.6.12;

interface IDAC {
    function memberLeave(address _creator, address _member) external returns (bool);
    function dismissDAC(address _creator) external returns (bool);
}
```

IDACRecorder.sol

```
pragma solidity 0.6.12;
```

```
interface IDACRecorder {
    enum Role { Creator, Member, None }
    enum DACState{ Active, Inactive }

    function checkUserInfo(address _user)
        external
        view
        returns (Role userRole, uint256 accPower, uint256 amount);
    function checkDACInfo(uint256 _dacId)
        external
        view
        returns (DACState state, uint256 userCount, uint256 accMetisPerShare, address creator, uint25
    function isCreator(address _user) external view returns (bool);
    function getUserAccPower(address _user) external view returns (uint256);
    function addCreator(address _user) external returns (bool);
    function removeCreator(address _user) external returns (bool);
    function addMember(uint256 _dacId, address _member) external returns (bool);
    function delMember(uint256 _dacId, address _member) external returns (bool);
    function updateCreatorInfo(
        address _user,
        uint256 _dacId,
        uint256 _DACMemberCount,
        uint256 _initialDACPower,
        uint256 _amount,
        uint256 _accMetisPerShare,
        bool _withdrawAll
    ) external returns (bool);
    function updateMemberInfo(
        address _user,
        uint256 _dacId,
        uint256 _DACMemberCount,
        uint256 _initialDACPower,
        uint256 _amount,
        bool _withdrawAll,
        bool _isDeposit
    ) external returns (bool);
    function subCreatorPower(uint256 _dacId, uint256 _amount) external returns (bool);
    function creatorOf(address _member) external returns (address);
    function setCreatorOf(address _creator, address _user) external;
    function totalWeight() external view returns (uint256);
    function MIN_MEMBER_COUNT() external view returns (uint256);
    function DAO_OPEN() external view returns (bool);
    function stakedMetis() external view returns (uint256);
    function sendRewardToVault(address _user, uint256 _amount) external returns (bool);
}
```

IDistributor.sol

```
pragma solidity 0.6.12;

interface IDistributor {
    function distribute(address _to, uint256 _amount) external returns (bool);
}
```

IMetisToken.sol

```
pragma solidity 0.6.12;

interface IMetisToken {
    function balanceOf(address account) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transfer(address recipient, uint256 amount) external returns (bool);
```

```
    function allowance(address owner, address spender) external view returns (uint256);
}
```

IMining.sol

```
pragma solidity 0.6.12;

interface IMining {
    function deposit(
        address _creator,
        address _user,
        uint256 _pid,
        uint256 _amount,
        uint256 _DACMemberCount,
        uint256 _initialDACPower,
        uint256 _dacId
    ) external returns (bool);

    function withdraw(address _creator, uint256 _pid, uint256 _amount, uint256 _dacId) external retur

    function dismissDAC(uint256 _dacId, uint256 _pid, address _creator) external returns (bool);

    function tokenToPid(address _token) external view returns (uint256);
}
```

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