Armors Labs

DekBox

Smart Contract Audit

DekBox Audit Summary

Project name : DekBox Contract

Project address: None

Code URL : None

Commit : None

Projct target : DekBox Contract Audit

Blockchain : Binance Smart Chain（BSC）

Test result : PASSED

Audit Info

Audit NO : 0X202104160016

Audit Team : Armors Labs

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

DekBox Audit

The DekBox team asked us to review and audit their DekBox 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
DekBox Audit	Rock, Hosea, Rushairer, Rico, David, Alice	1.0.0	2021-04-16

Audit results

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the DekBox 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 18 months from the date of output.

(Statement: Armors Labs reports only on facts that have occurred or existed before this report is issued and assumes corresponding responsibilities. Armors Labs is not able to determine the security of its smart contracts and is not responsible for any subsequent or existing facts after this report is issued. The security audit analysis and other content of this report are only based on the documents and information provided by the information provider to Armors Labs at the time of issuance of this report (" information provided " for short). 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.)

Audited target file

file	md5
Ctoken.sol	aa9dd35ec3beed8937debc017dbf0e81
MastChef.sol	da48d6492964a6f8088ab3d295bc0ab9
Timelock.sol	5bdef10b5a3ca366393e745932805354

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

Vulnerability	status
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe

Contract file

Ctoken.sol

```
pragma solidity ^0.5.16;
pragma experimental ABIEncoderV2;

contract Comp {
    /// @notice EIP-20 token name for this token
    string public constant name = "DekBox";

    /// @notice EIP-20 token symbol for this token
    string public constant symbol = "DEK";

    /// @notice EIP-20 token decimals for this token
    uint8 public constant decimals = 18;

    /// @notice Total number of tokens in circulation
    uint256 public constant totalSupply = 90000000e18; // 1 billion comp

    /// @notice Allowance amounts on behalf of others
    mapping(address => mapping(address => uint96)) internal allowances;

    /// @notice Official record of token balances for each account
    mapping(address => uint96) internal balances;

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

    /// @notice A checkpoint for marking number of votes from a given block
    struct Checkpoint {
        uint32 fromBlock;
        uint96 votes;
    }

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

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

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

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

    /// @notice A record of states for signing / validating signatures
    mapping(address => uint256) public nonces;

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

    /// @notice An event thats emitted when a delegate account's vote balance changes
    event DelegateVotesChanged(
        address indexed delegate,
        uint256 previousBalance,
        uint256 newBalance
    );

    /// @notice The standard EIP-20 transfer event
    event Transfer(address indexed from, address indexed to, uint256 amount);

    /// @notice The standard EIP-20 approval event
    event Approval(
        address indexed owner,
        address indexed spender,
        uint256 amount
    );

    /**
     * @notice Construct a new comp token
     * @param account The initial account to grant all the tokens
     */
    constructor(address account) public {
        balances[account] = uint96(totalSupply);
        emit Transfer(address(0), account, totalSupply);
    }

    /**
     * @notice Get the number of tokens `spender` is approved to spend on behalf of `account`
     * @param account The address of the account holding the funds
     * @param spender The address of the account spending the funds
     * @return The number of tokens approved
     */
    function allowance(address account, address spender)
        external
        view
        returns (uint256)
    {
        return allowances[account][spender];
    }

    /**
     * @notice Approve `spender` to transfer up to `amount` from `src`
     * @dev This will overwrite the approval amount for `spender`
     *  and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
     * @param spender The address of the account which may transfer tokens
     * @param rawAmount The number of tokens that are approved (2^256-1 means infinite)
     * @return Whether or not the approval succeeded
     */
    function approve(address spender, uint256 rawAmount)
        external
        returns (bool)
    {
        uint96 amount;
        if (rawAmount == uint256(-1)) {
            amount = uint96(-1);
        } else {
            amount = safe96(rawAmount, "Comp::approve: amount exceeds 96 bits");
        }

        allowances[msg.sender][spender] = amount;

        emit Approval(msg.sender, spender, amount);
        return true;
    }

    /**
     * @notice Get the number of tokens held by the `account`
     * @param account The address of the account to get the balance of
     * @return The number of tokens held
     */
    function balanceOf(address account) external view returns (uint256) {
        return balances[account];
    }

    /**
     * @notice Transfer `amount` tokens from `msg.sender` to `dst`
     * @param dst The address of the destination account
     * @param rawAmount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transfer(address dst, uint256 rawAmount) external returns (bool) {
        uint96 amount =
            safe96(rawAmount, "Comp::transfer: amount exceeds 96 bits");
        _transferTokens(msg.sender, dst, amount);
        return true;
    }

    /**
     * @notice Transfer `amount` tokens from `src` to `dst`
     * @param src The address of the source account
     * @param dst The address of the destination account
     * @param rawAmount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transferFrom(
        address src,
        address dst,
        uint256 rawAmount
    ) external returns (bool) {
        address spender = msg.sender;
        uint96 spenderAllowance = allowances[src][spender];
        uint96 amount =
            safe96(rawAmount, "Comp::approve: amount exceeds 96 bits");

        if (spender != src && spenderAllowance != uint96(-1)) {
            uint96 newAllowance =
                sub96(
                    spenderAllowance,
                    amount,
                    "Comp::transferFrom: transfer amount exceeds spender allowance"
                );
            allowances[src][spender] = newAllowance;

            emit Approval(src, spender, newAllowance);
        }

        _transferTokens(src, dst, amount);
        return true;
    }

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

```
/**
 * @notice Delegates votes from signatory to `delegatee`
 * @param delegatee The address to delegate votes to
 * @param nonce The contract state required to match the signature
 * @param expiry The time at which to expire the signature
 * @param v The recovery byte of the signature
 * @param r Half of the ECDSA signature pair
 * @param s Half of the ECDSA signature pair
 */
function delegateBySig(
    address delegatee,
    uint256 nonce,
    uint256 expiry,
    uint8 v,
    bytes32 r,
    bytes32 s
) public {
    bytes32 domainSeparator =
        keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name)),
                getChainId(),
                address(this)
            )
        );
    bytes32 structHash =
        keccak256(
            abi.encode(DELEGATION_TYPEHASH, delegatee, nonce, expiry)
        );
    bytes32 digest =
        keccak256(
            abi.encodePacked("\x19\x01", domainSeparator, structHash)
        );
    address signatory = ecrecover(digest, v, r, s);
    require(
        signatory != address(0),
        "Comp::delegateBySig: invalid signature"
    );
    require(
        nonce == nonces[signatory]++,
        "Comp::delegateBySig: invalid nonce"
    );
    require(now <= expiry, "Comp::delegateBySig: signature expired");
    return _delegate(signatory, delegatee);
}

/**
 * @notice Gets the current votes balance for `account`
 * @param account The address to get votes balance
 * @return The number of current votes for `account`
 */
function getCurrentVotes(address account) external view returns (uint96) {
    uint32 nCheckpoints = numCheckpoints[account];
    return
        nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
}

/**
 * @notice Determine the prior number of votes for an account as of a block number
 * @dev Block number must be a finalized block or else this function will revert to prevent misin
 * @param account The address of the account to check
 * @param blockNumber The block number to get the vote balance at
 * @return The number of votes the account had as of the given block
 */
function getPriorVotes(address account, uint256 blockNumber)
```

```
        public
        view
        returns (uint96)
    {
        require(
            blockNumber < block.number,
            "Comp::getPriorVotes: not yet determined"
        );

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

        // First check most recent balance
        if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
            return checkpoints[account][nCheckpoints - 1].votes;
        }

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

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

    function _delegate(address delegator, address delegatee) internal {
        address currentDelegate = delegates[delegator];
        uint96 delegatorBalance = balances[delegator];
        delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }

    function _transferTokens(
        address src,
        address dst,
        uint96 amount
    ) internal {
        require(
            src != address(0),
            "Comp::_transferTokens: cannot transfer from the zero address"
        );
        require(
            dst != address(0),
            "Comp::_transferTokens: cannot transfer to the zero address"
        );

        balances[src] = sub96(
            balances[src],
            amount,
            "Comp::_transferTokens: transfer amount exceeds balance"
        );
        balances[dst] = add96(
            balances[dst],
            amount,
            "Comp::_transferTokens: transfer amount overflows"
        );
        emit Transfer(src, dst, amount);

        _moveDelegates(delegates[src], delegates[dst], amount);
    }

    function _moveDelegates(
        address srcRep,
        address dstRep,
        uint96 amount
    ) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint96 srcRepOld =
                    srcRepNum > 0
                        ? checkpoints[srcRep][srcRepNum - 1].votes
                        : 0;
                uint96 srcRepNew =
                    sub96(
                        srcRepOld,
                        amount,
                        "Comp::_moveVotes: vote amount underflows"
                    );
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint96 dstRepOld =
                    dstRepNum > 0
                        ? checkpoints[dstRep][dstRepNum - 1].votes
                        : 0;
                uint96 dstRepNew =
                    add96(
                        dstRepOld,
                        amount,
                        "Comp::_moveVotes: vote amount overflows"
                    );
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }

    function _writeCheckpoint(
        address delegatee,
        uint32 nCheckpoints,
        uint96 oldVotes,
        uint96 newVotes
    ) internal {
        uint32 blockNumber =
            safe32(
                block.number,
                "Comp::_writeCheckpoint: block number exceeds 32 bits"
            );

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

        emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
    }

    function safe32(uint256 n, string memory errorMessage)
        internal
        pure
        returns (uint32)
    {
        require(n < 2**32, errorMessage);
        return uint32(n);
    }

    function safe96(uint256 n, string memory errorMessage)
        internal
        pure
        returns (uint96)
    {
        require(n < 2**96, errorMessage);
        return uint96(n);
    }

    function add96(
        uint96 a,
        uint96 b,
        string memory errorMessage
    ) internal pure returns (uint96) {
        uint96 c = a + b;
        require(c >= a, errorMessage);
        return c;
    }

    function sub96(
        uint96 a,
        uint96 b,
        string memory errorMessage
    ) internal pure returns (uint96) {
        require(b <= a, errorMessage);
        return a - b;
    }

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

MastChef.sol

```
pragma solidity ^0.6.0;
```

```
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;
    }
}

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;
    }
}

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 message on
     * 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 multiplicarion of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     *
     * - Multiplicarion 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: multiplicarion 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 message 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, 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 modulo),
     * 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 modulo),
     * 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;
    }
}

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 contract.
     *
     * Among others, `isContract` will return false for the following
     * types of addresses:
     *
     *  - an externally-owned account
     *  - a contract in construction
     *  - an address where a contract will be created
     *  - an address where a contract lived, but was destroyed
     * ====
     */
    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-checks-effects
     */
    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 level `call`. A
     * plain`call` is an unsafe replacement for a function call: use this
     * 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 expected return value,
     * use https://solidity.readthedocs.io/en/latest/units-and-global-variables.html?highlight=abi.de
     *
     * 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 least `value`.
     * - 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-}[`functionCallWithValu
     * 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);
            }
        }
    }
}

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 indicaring whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, uint256 amount) external returns (bool);

    /**
     * @dev Returns the remaining number of tokens that `spender` will be
     * 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 caller's tokens.
     *
     * Returns a boolean value indicaring whether the operation succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this method brings the risk
     * that someone may use both the old and the new allowance by unfortunate
     * 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 indicaring whether the operation succeeded.
     *
     * 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 an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);
}

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 Deprecared. 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");
        }
    }
}

interface IMigratorChef {
    // Perform LP token migration from legacy UniswapV2 to Swap.
    // Take the current LP token address and return the new LP token address.
    // Migrator should have full access to the caller's LP token.
    // Return the new LP token address.
    //
    // XXX Migrator must have allowance access to UniswapV2 LP tokens.
    // Swap must mint EXACTLY the same amount of Swap LP tokens or
    // else something bad will happen. Traditional UniswapV2 does not
    // do that so be careful!
    function migrate(IERC20 token) external returns (IERC20);
}

contract MasterChef is Ownable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    // Info of each user.
    struct UserInfo {
        uint256 amount;     // How many LP tokens the user has provided.
        uint256 rewardDebt; // Reward debt. See explanation below.
        //
        // We do some fancy math here. Basically, any point in time, the amount of DEKs
        // entitled to a user but is pending to be distributed is:
        //
        //   pending reward = (user.amount * pool.accDekPerShare) - user.rewardDebt
        //
        // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
        //   1. The pool's `accDekPerShare` (and `lastRewardBlock`) gets updated.
        //   2. User receives the pending reward sent to his/her address.
        //   3. User's `amount` gets updated.
        //   4. User's `rewardDebt` gets updated.
    }

    // Info of each pool.
    struct PoolInfo {
        IERC20 lpToken;           // Address of LP token contract.
        uint256 allocPoint;       // How many allocarion points assigned to this pool. DEKs to distri
        uint256 lastRewardBlock;  // Last block number that DEKs distribution occurs.
        uint256 accDekPerShare; // Accumulated DEKs per share, times 1e12. See below.
    }

    // The DEK TOKEN!
    IERC20 public DEK;
    // DEK tokens created per block.
    uint256 public dekPerBlock;
    // The migrator contract. It has a lot of power. Can only be set through governance (owner).
    IMigratorChef public migrator;

    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each user that stakes LP tokens.
    mapping (uint256 => mapping (address => UserInfo)) public userInfo;
    // Total allocarion poitns. Must be the sum of all allocarion points in all pools.
    uint256 public totalAllocPoint = 0;
    // The block number when dek mining starts.
    uint256 public startBlock;
    uint256 public mintDEK = 0;

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

    constructor(
        IERC20 _dek,
        uint256 _dekPerBlock,
        uint256 _startBlock
    ) public {
        DEK = _dek;
        dekPerBlock = _dekPerBlock;
        startBlock = _startBlock;
    }

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

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accDekPerShare: 0
        }));
    }

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

    //set dekPerBlock
    function setPerParam(uint256 _amount, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        dekPerBlock = _amount;
    }

    // Set the migrator contract. Can only be called by the owner.
    function setMigrator(IMigratorChef _migrator) public onlyOwner {
        migrator = _migrator;
    }

    // Migrate lp token to another lp contract. Can be called by anyone. We trust that migrator contr
    function migrate(uint256 _pid) public {
        require(address(migrator) != address(0), "migrate: no migrator");
        PoolInfo storage pool = poolInfo[_pid];
        IERC20 lpToken = pool.lpToken;
        uint256 bal = lpToken.balanceOf(address(this));
        lpToken.safeApprove(address(migrator), bal);
        IERC20 newLpToken = migrator.migrate(lpToken);
        require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
        pool.lpToken = newLpToken;
    }

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

    // View function to see pending Deks on frontend.
    function pendingDek(uint256 _pid, address _user) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accDekPerShare = pool.accDekPerShare;
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (block.number > pool.lastRewardBlock && lpSupply != 0) {
            uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
            uint256 dekReward = multiplier.mul(dekPerBlock).mul(pool.allocPoint).div(totalAllocPoint)
            accDekPerShare = accDekPerShare.add(dekReward.mul(1e12).div(lpSupply));
        }
        return user.amount.mul(accDekPerShare).div(1e12).sub(user.rewardDebt);
    }

    // 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.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 dekReward = multiplier.mul(dekPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
        // DEK.mint(address(this), dekReward);
        mintDEK = mintDEK.add(dekReward);
        require(DEK.balanceOf(address(this)) >= mintDEK, "not enough DEK in contract");
        pool.accDekPerShare = pool.accDekPerShare.add(dekReward.mul(1e12).div(lpSupply));
        pool.lastRewardBlock = block.number;
    }

    // Deposit LP tokens to MasterChef for Dek allocarion.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accDekPerShare).div(1e12).sub(user.rewardDebt);
            safeDekTransfer(msg.sender, pending);
        }
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accDekPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }
    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 pending = user.amount.mul(pool.accDekPerShare).div(1e12).sub(user.rewardDebt);
        safeDekTransfer(msg.sender, pending);
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accDekPerShare).div(1e12);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit Withdraw(msg.sender, _pid, _amount);
    }

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

    // Safe Dek transfer function, just in case if rounding error causes pool to not have enough Deks
    function safeDekTransfer(address _to, uint256 _amount) internal {
        uint256 dekBal = DEK.balanceOf(address(this));
        if (_amount > dekBal) {
            mintDEK.sub(dekBal);
            DEK.safeTransfer(_to, dekBal);
        } else {
            mintDEK.sub(_amount);
            DEK.safeTransfer(_to, _amount);
        }
    }
}
```

Timelock.sol

```
// File: contracts/SafeMath.sol

pragma solidity ^0.5.16;

// From https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/Math.sol
// Subject to the MIT license.

/**
 * @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 an
 * 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 entire
 * 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 addition of two unsigned integers, reverting with custom message on overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     * - Addition cannot overflow.
     */
    function add(
        uint256 a,
        uint256 b,
        string memory errorMessage
    ) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, errorMessage);

        return c;
    }

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

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on underf
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     * - Subtraction cannot underflow.
     */
    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 multiplication of two unsigned integers, reverting on overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     * - Multiplication cannot overflow.
     */
    function mul(
        uint256 a,
        uint256 b,
        string memory errorMessage
    ) 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, errorMessage);

        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 message 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,
        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 modulo),
     * 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 modulo),
     * 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;
    }
}

// File: contracts/Timelock.sol

pragma solidity ^0.5.16;

contract Timelock {
    using SafeMath for uint256;

    event NewAdmin(address indexed newAdmin);
    event NewPendingAdmin(address indexed newPendingAdmin);
    event NewDelay(uint256 indexed newDelay);
    event CancelTransaction(
        bytes32 indexed txHash,
        address indexed target,
        uint256 value,
        string signature,
        bytes data,
        uint256 eta
    );
    event ExecuteTransaction(
        bytes32 indexed txHash,
        address indexed target,
        uint256 value,
        string signature,
        bytes data,
        uint256 eta
    );
    event QueueTransaction(
        bytes32 indexed txHash,
        address indexed target,
        uint256 value,
        string signature,
        bytes data,
        uint256 eta
    );

    uint256 public constant GRACE_PERIOD = 5000 seconds;
    uint256 public constant MINIMUM_DELAY = 60 seconds;
    uint256 public constant MAXIMUM_DELAY = 500 seconds;

    address public admin;
    address public pendingAdmin;
    uint256 public delay;
    bool public admin_initialized;

    mapping(bytes32 => bool) public queuedTransactions;

    constructor(address admin_, uint256 delay_) public {
        require(
            delay_ >= MINIMUM_DELAY,
            "Timelock::constructor: Delay must exceed minimum delay."
        );
        require(
            delay_ <= MAXIMUM_DELAY,
            "Timelock::constructor: Delay must not exceed maximum delay."
        );

        admin = admin_;
        delay = delay_;
        admin_initialized = false;
    }

    function() external payable {}

    function setDelay(uint256 delay_) public {
        require(
            msg.sender == address(this),
            "Timelock::setDelay: Call must come from Timelock."
        );
        require(
            delay_ >= MINIMUM_DELAY,
            "Timelock::setDelay: Delay must exceed minimum delay."
        );
        require(
            delay_ <= MAXIMUM_DELAY,
            "Timelock::setDelay: Delay must not exceed maximum delay."
        );
        delay = delay_;

        emit NewDelay(delay);
    }

    function acceptAdmin() public {
        require(
            msg.sender == pendingAdmin,
            "Timelock::acceptAdmin: Call must come from pendingAdmin."
        );
        admin = msg.sender;
        pendingAdmin = address(0);

        emit NewAdmin(admin);
    }

    function setPendingAdmin(address pendingAdmin_) public {
        // allows one time setting of admin for deployment purposes
        if (admin_initialized) {
            require(
                msg.sender == address(this),
                "Timelock::setPendingAdmin: Call must come from Timelock."
            );
        } else {
            require(
                msg.sender == admin,
                "Timelock::setPendingAdmin: First call must come from admin."
            );
            admin_initialized = true;
        }
        pendingAdmin = pendingAdmin_;

        emit NewPendingAdmin(pendingAdmin);
    }

    function queueTransaction(
        address target,
        uint256 value,
        string memory signature,
        bytes memory data,
        uint256 eta
    ) public returns (bytes32) {
        require(
            msg.sender == admin,
            "Timelock::queueTransaction: Call must come from admin."
        );
        require(
            eta >= getBlockTimestamp().add(delay),
            "Timelock::queueTransaction: Estimated execution block must satisfy delay."
        );

        bytes32 txHash =
            keccak256(abi.encode(target, value, signature, data, eta));
        queuedTransactions[txHash] = true;

        emit QueueTransaction(txHash, target, value, signature, data, eta);
        return txHash;
    }

    function cancelTransaction(
        address target,
        uint256 value,
        string memory signature,
        bytes memory data,
        uint256 eta
    ) public {
        require(
            msg.sender == admin,
            "Timelock::cancelTransaction: Call must come from admin."
        );

        bytes32 txHash =
            keccak256(abi.encode(target, value, signature, data, eta));
        queuedTransactions[txHash] = false;

        emit CancelTransaction(txHash, target, value, signature, data, eta);
    }

    function executeTransaction(
        address target,
        uint256 value,
        string memory signature,
        bytes memory data,
        uint256 eta
    ) public payable returns (bytes memory) {
        require(
            msg.sender == admin,
            "Timelock::executeTransaction: Call must come from admin."
        );

        bytes32 txHash =
            keccak256(abi.encode(target, value, signature, data, eta));
        require(
            queuedTransactions[txHash],
            "Timelock::executeTransaction: Transaction hasn't been queued."
        );
        require(
            getBlockTimestamp() >= eta,
            "Timelock::executeTransaction: Transaction hasn't surpassed time lock."
        );
        require(
            getBlockTimestamp() <= eta.add(GRACE_PERIOD),
            "Timelock::executeTransaction: Transaction is stale."
        );

        queuedTransactions[txHash] = false;

        bytes memory callData;

        if (bytes(signature).length == 0) {
            callData = data;
        } else {
            callData = abi.encodePacked(
                bytes4(keccak256(bytes(signature))),
                data
            );
        }

        // solium-disable-next-line security/no-call-value
        (bool success, bytes memory returnData) =
            target.call.value(value)(callData);
        require(
            success,
            "Timelock::executeTransaction: Transaction execution reverted."
        );

        emit ExecuteTransaction(txHash, target, value, signature, data, eta);

        return returnData;
    }

    function getBlockTimestamp() internal view returns (uint256) {
        // solium-disable-next-line security/no-block-members
        return block.timestamp;
    }
}
```

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.

armors.io

contact@armors.io