kaspersky

Smart Contract Code Review and Security Analysis Report

30.09.2019

kaspersky

Smart Contract Code Review and Security Analysis Report for Revain

This document contains confidential information about intellectual property of the customer, as well as information about potential vulnerabilities and methods of their exploitation. This report may be published upon agreement with the Customer.

Introduction

Kaspersky was contracted by **Revain** to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of Customer`s smart contract and its code review conducted on **5.08.2019-12.08.2019**. The Customer has updated the code, the second review was conducted on **23.09.2019**.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, merely an assessment of its logic and implementation. The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding to several independent audits and a public bug bounty program to ensure the security of the smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

Executive Summary

Initial code review revealed seven issues of different severity levels with Medium as maximum. Revain team has addressed all the issues found and provided updated source code and project technical documentation.

The reviewed source codes are well crafted and follow common security practices.

Audit team has found no security issues during an audit. Audit report contains all necessary information related to them.

Because no issues were found, the Code Review Verdict assigned is **green**.

kaspersky

Scope

On 02.08.2019 contract source codes were obtained from Revain, including (with SHA256):

src_complete.sol	47B0B47400FB4BB0B74A4C78B0303B6418D3C60A47E7D9D7868117289661179A
sign.js	B30EEFEF5A4C4BB8F7A2B29122DA8CC2599D0E31F46C514D45975BAEB722D889
callBackSelector.sol	303098CFDC1641865ED83412C821B7FD744C0813BE0A08BDCD44BE0C7F542BA6

On 23.09.2019 updated contract source codes and technical documentation were obtained from Revain, including (with SHA256):

RBN_source.sol	DBB4A6BFCDE69784B114219327F42F3DF84DD2269AFFCC8CA97975632F47CB34
sign.js	50DF51FC28ACFE0006B560223F2E81AB6E6293F98883FD5BE9B58247B1BD3FF7
callbackSelector.sol	DB320B44F0F127E18AF6A328B926F6CEBD62F1D0A9648F9AFE9BA20DB33193B0
RBN Description.pdf	CCDD9BCBB3402BFBA79EA7D35FE467A60C9C9FC1C52C06A6A88AA182F1173E19

Review Methodology

Throughout the review process, care is taken to ensure that the token contract:

- Implements and adheres to existing ERC-20 Token standard appropriately and effectively
- Documentation and code comments match logic and behavior
- Distributes tokens in a manner that described in corresponding Token Sale T&C (the smart contract is compliant with the requirement of Customer logic, matching the initial constant values etc.)
- Follows best practices in efficient use of gas, without unnecessary waste
- Uses methods safe from reentrance attacks
- Is not affected by known vulnerabilities

To do so our team of experts review the code line-by-line documenting any issues as they are discovered.

We are scanning this smart contract for commonly known and more specific vulnerabilities. Here are some of the vulnerabilities that are considered (the full list includes them but is not limited to them):

- Reentrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Byte array vulnerabilities
- Style guide violation
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

The static analysis portion of our audit is performed using a series of automated tools, purposefully designed to test the security of the contract, such as Remix, Oyente, Solhint.

kaspersky

All the issues are divided into several risk levels:

● **Informational** - The issue has no impact on the contract's ability to operate (code style violations and info statements, can't affect smart contract execution and can be ignored)

● **Low** - The issue has minimal impact on the contract's ability to operate (mostly related to outdated, unused etc. code snippets)

● **Medium** - The issue affects the ability of the contract to operate in a way that doesn't significantly hinder its behavior

● **High** - High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions

● **Critical** - The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss

A Code Review Verdict is assigned, which indicates a qualitative score for the Smart Contract source code. Verdict is represented by color mark: red, yellow or green, where

- **red** indicates the lowest possible source code quality, one or more high/critical issues found
- **yellow** indicates moderate source code quality, no high/critical issues found and one or more medium issues found
- **green** indicates the highest possible source code quality, no medium/high/critical issues found

Issues List

During Review no issues were found.

kaspersky

Appendix A

Reviewed contracts' source code listing can be found below.

sign.js

```
const msgHash = "0x…" // put your messageHash here

const parseSignature = (signature) => {
    var rec = {messageHash: msgHash, signature: signature ,v: '',r: '',s: ''};
    var sign = signature.slice(2)
    rec.r = "0x" + sign.slice(0,64)
    rec.s = "0x" + sign.slice(64, 128)
    rec.v = "0x" + sign.slice(128, 130)
    rec.v = web3.utils.hexToNumber(rec.v)
    console.log(rec)

}

const sign = (address) => {
    web3.eth.personal.sign(web3.utils.toHex(msgHash), address).then(parseSignature)
}

web3.eth.getCoinbase().then(sign)
```

callBackSelector.sol

```
pragma solidity ^0.5.10;

/**
  * @title contract for generating HEX pointer
  * for functions.
  *
  * @dev this contract is a tool meant to be used
  * on local JavaScript VM.
  *
  */
contract callbackSelector {

    /**
      * @notice function which returns function HEX pointer (callbackSelector)
      *
      * @param _function function name with parameter types. Case and whitespace sensitive.
      *
      * @dev example: `function get(string memory _function)`
      *     _function: `get(string)`
      *       result: `0x693ec85e`
      */
    function get(string memory _function) public pure returns (bytes4) {
        return bytes4(keccak256(abi.encodePacked(_function)));
    }
}
```

kaspersky

RBN_source.sol

```
pragma solidity ^0.5.10;

/** @title  A contract for generating unique identifiers
  *
  * @notice  A contract that provides an identifier generation scheme,
  * guaranteeing uniqueness across all contracts that inherit from it,
  * as well as the unpredictability of future identifiers.
  *
  * @dev  This contract is intended to be inherited by any contract that
  * implements the callback software pattern for cooperative custodianship.
  *
*/
contract LockRequestable {

    // MEMBERS
    /// @notice  the count of all invocations of `generateLockId`.
    uint256 public lockRequestCount;

    // CONSTRUCTOR
    constructor() public {
        lockRequestCount = 0;
    }

    // FUNCTIONS
    /** @notice  Returns a fresh unique identifier.
      *
      * @dev the generation scheme uses three components.
      * First, the blockhash of the previous block.
      * Second, the deployed address.
      * Third, the next value of the counter.
      * This ensures that identifiers are unique across all contracts
      * following this scheme, and that future identifiers are
      * unpredictable.
      *
      * @return a 32-byte unique identifier.
    */
    function generateLockId() internal returns (bytes32 lockId) {
        return keccak256(abi.encodePacked(blockhash(block.number - 1), address(this), ++lockRequestCount));
    }
}

contract ERC20Interface {

    // METHODS

    // NOTE:
    //   public getter functions are not currently recognised as an
    //   implementation of the matching abstract function by the compiler.

    // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#name
    // function name() public view returns (string);

    // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#symbol
    // function symbol() public view returns (string);

    // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#totalsupply
    // function decimals() public view returns (uint8);
```

```
    // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#totalsupply
    function totalSupply() public view returns (uint256);

    // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#balanceof
    function balanceOf(address _owner) public view returns (uint256 balance);

    // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#transfer
    function transfer(address _to, uint256 _value) public returns (bool success);

    // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#transferfrom
    function transferFrom(address _from, address _to, uint256 _value) public returns (bool success);

    // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#approve
    function approve(address _spender, uint256 _value) public returns (bool success);

    // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#allowance
    function allowance(address _owner, address _spender) public view returns (uint256 remaining);

    // EVENTS
    // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#transfer-1
    event Transfer(address indexed _from, address indexed _to, uint256 _value);

    // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#approval
    event Approval(address indexed _owner, address indexed _spender, uint256 _value);
}

/** @title  A dual control contract.
  *
  * @notice  A general-purpose contract that implements dual control over
  * co-operating contracts through a callback mechanism.
  *
  * @dev  This contract implements dual control through a 2-of-N
  * threshold multi-signature scheme. The contract recognizes a set of N signers,
  * and will unlock requests with signatures from any distinct pair of them.
  * This contract signals the unlocking through a co-operative callback
  * scheme.
  * This contract also provides time lock and revocation features.
  * Requests made by a 'primary' account have a default time lock applied.
  * All other requests must pay a 1 ETH stake and have an extended time lock
  * applied.
  * A request that is completed will prevent all previous pending requests
  * that share the same callback from being completed: this is the
  * revocation feature.
  *
  */
contract Custodian {

    // TYPES
    /** @dev  The `Request` struct stores a pending unlocking.
      * `callbackAddress` and `callbackSelector` are the data required to
      * make a callback. The custodian completes the process by
      * calling `callbackAddress.call(callbackSelector, lockId)`, which
      * signals to the contract co-operating with the Custodian that
      * the 2-of-N signatures have been provided and verified.
      */
    struct Request {
        bytes32 lockId;
        bytes4 callbackSelector;  // bytes4 and address can be packed into 1 word
        address callbackAddress;
        uint256 idx;
```

```
    uint256 timestamp;
    bool extended;
}

// EVENTS
/// @dev  Emitted by successful `requestUnlock` calls.
event Requested(
    bytes32 _lockId,
    address _callbackAddress,
    bytes4 _callbackSelector,
    uint256 _nonce,
    address _whitelistedAddress,
    bytes32 _requestMsgHash,
    uint256 _timeLockExpiry
);

/// @dev  Emitted by `completeUnlock` calls on requests in the time-locked state.
event TimeLocked(
    uint256 _timeLockExpiry,
    bytes32 _requestMsgHash
);

/// @dev  Emitted by successful `completeUnlock` calls.
event Completed(
    bytes32 _lockId,
    bytes32 _requestMsgHash,
    address _signer1,
    address _signer2
);

/// @dev  Emitted by `completeUnlock` calls where the callback failed.
event Failed(
    bytes32 _lockId,
    bytes32 _requestMsgHash,
    address _signer1,
    address _signer2
);

/// @dev  Emitted by successful `extendRequestTimeLock` calls.
event TimeLockExtended(
    uint256 _timeLockExpiry,
    bytes32 _requestMsgHash
);

 // MEMBERS
/** @dev  The count of all requests.
  * This value is used as a nonce, incorporated into the request hash.
  */
uint256 public requestCount;

/// @dev  The set of signers: signatures from two signers unlock a pending request.
mapping (address => bool) public signerSet;

/// @dev  The map of request hashes to pending requests.
mapping (bytes32 => Request) public requestMap;

/// @dev  The map of callback addresses to callback selectors to request indexes.
mapping (address => mapping (bytes4 => uint256)) public lastCompletedIdxs;

/** @dev  The default period (in seconds) to time-lock requests.
```

```
  * All requests will be subject to this default time lock, and the duration
  * is fixed at contract creation.
  */
uint256 public defaultTimeLock;

/** @dev  The extended period (in seconds) to time-lock requests.
  * Requests not from the primary account are subject to this time lock.
  * The primary account may also elect to extend the time lock on requests
  * that originally received the default.
  */
uint256 public extendedTimeLock;


/// @dev  The primary account is the privileged account for making requests.
address public primary;


// CONSTRUCTOR
constructor(
    address[] memory _signers,
    uint256 _defaultTimeLock,
    uint256 _extendedTimeLock,
    address _primary
)
    public
{
    // check for at least two `_signers`
    require(_signers.length >= 2, "at least two `_signers`");


     // validate time lock params
    require(_defaultTimeLock <= _extendedTimeLock, "valid timelock params");
    defaultTimeLock = _defaultTimeLock;
    extendedTimeLock = _extendedTimeLock;


    primary = _primary;


    // explicitly initialize `requestCount` to zero
    requestCount = 0;
    // turn the array into a set
    for (uint i = 0; i < _signers.length; i++) {
        // no zero addresses or duplicates
        require(_signers[i] != address(0) && !signerSet[_signers[i]], "no zero addresses or duplicates");
```

```
        signerSet[_signers[i]] = true;
    }
}

// MODIFIERS
modifier onlyPrimary {
    require(msg.sender == primary, "only primary");
    _;
}

 modifier onlySigner {
    require(signerSet[msg.sender], "only signer");
    _;
}

// METHODS
/** @notice  Requests an unlocking with a lock identifier and a callback.
  *
  * @dev  If called by an account other than the primary a 1 ETH stake
  * must be paid. When the request is unlocked stake will be transferred to the message sender.
  * This is an anti-spam measure. As well as the callback
  * and the lock identifier parameters a 'whitelisted address' is required
  * for compatibility with existing signature schemes.
  *
  * @param  _lockId  The identifier of a pending request in a co-operating contract.
  * @param  _callbackAddress  The address of a co-operating contract.
  * @param  _callbackSelector  The function selector of a function within
  * the co-operating contract at address `_callbackAddress`.
  * @param  _whitelistedAddress  An address whitelisted in existing
  * offline control protocols.
  *
  * @return  requestMsgHash  The hash of a request message to be signed.
*/
function requestUnlock(
    bytes32 _lockId,
    address _callbackAddress,
    bytes4 _callbackSelector,
```

```
    address _whitelistedAddress
)
    public
    payable
    returns (bytes32 requestMsgHash)
{
    require(msg.sender == primary || msg.value >= 1 ether, "sender is primary or stake is paid");
    // disallow using a zero value for the callback address
    require(_callbackAddress != address(0), "no zero value for callback address");
    uint256 requestIdx = ++requestCount;
    // compute a nonce value
    // - the blockhash prevents prediction of future nonces
    // - the address of this contract prevents conflicts with co-operating contracts using this scheme
    // - the counter prevents conflicts arising from multiple txs within the same block
    uint256 nonce = uint256(keccak256(abi.encodePacked(blockhash(block.number - 1), address(this), requestIdx)));

    requestMsgHash = keccak256(
        abi.encodePacked(
            nonce,
            _whitelistedAddress,
            uint256(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)
        )
    );
    requestMap[requestMsgHash] = Request({
        lockId: _lockId,
        callbackSelector: _callbackSelector,
        callbackAddress: _callbackAddress,
        idx: requestIdx,
        timestamp: block.timestamp,
        extended: false
    });

    // compute the expiry time
    uint256 timeLockExpiry = block.timestamp;
    if (msg.sender == primary) {
        timeLockExpiry += defaultTimeLock;
    } else {
```

```
            timeLockExpiry += extendedTimeLock;
            // any sender that is not the creator will get the extended time lock
            requestMap[requestMsgHash].extended = true;
        }
        emit Requested(_lockId, _callbackAddress, _callbackSelector, nonce, _whitelistedAddress, requestMsgHash,
timeLockExpiry);
    }
    /** @notice  Completes a pending unlocking with two signatures.
     *
     * @dev  Given a request message hash as two signatures of it from
     * two distinct signers in the signer set, this function completes the
     * unlocking of the pending request by executing the callback.
     *
     * @param  _requestMsgHash  The request message hash of a pending request.
     * @param  _recoveryByte1  The public key recovery byte (27 or 28)
     * @param  _ecdsaR1  The R component of an ECDSA signature (R, S) pair
     * @param  _ecdsaS1  The S component of an ECDSA signature (R, S) pair
     * @param  _recoveryByte2  The public key recovery byte (27 or 28)
     * @param  _ecdsaR2  The R component of an ECDSA signature (R, S) pair
     * @param  _ecdsaS2  The S component of an ECDSA signature (R, S) pair
     *
     * @return  success  True if the callback successfully executed.
    */
    function completeUnlock(
        bytes32 _requestMsgHash,
        uint8 _recoveryByte1, bytes32 _ecdsaR1, bytes32 _ecdsaS1,
        uint8 _recoveryByte2, bytes32 _ecdsaR2, bytes32 _ecdsaS2
    )
        public
        onlySigner
        returns (bool success)
    {
        Request storage request = requestMap[_requestMsgHash];

        // copy storage to locals before `delete`
        bytes32 lockId = request.lockId;
        address callbackAddress = request.callbackAddress;
```

```
bytes4 callbackSelector = request.callbackSelector;

// failing case of the lookup if the callback address is zero
require(callbackAddress != address(0), "no zero value for callback address");

// reject confirms of earlier withdrawals buried under later confirmed withdrawals
require(request.idx > lastCompletedIdxs[callbackAddress][callbackSelector],
"reject confirms of earlier withdrawals buried under later confirmed withdrawals");

address signer1 = ecrecover(
    keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", _requestMsgHash)),
    _recoveryByte1,
    _ecdsaR1,
    _ecdsaS1
);
require(signerSet[signer1], "signer is set");

address signer2 = ecrecover(
    keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", _requestMsgHash)),
    _recoveryByte2,
    _ecdsaR2,
    _ecdsaS2
);
require(signerSet[signer2], "signer is set");
require(signer1 != signer2, "signers are different");

if (request.extended && ((block.timestamp - request.timestamp) < extendedTimeLock)) {
    emit TimeLocked(request.timestamp + extendedTimeLock, _requestMsgHash);
    return false;
} else if ((block.timestamp - request.timestamp) < defaultTimeLock) {
    emit TimeLocked(request.timestamp + defaultTimeLock, _requestMsgHash);
    return false;
} else {
    if (address(this).balance > 0) {
        // reward sender with anti-spam payments
        msg.sender.transfer(address(this).balance);
    }
```

```
        // raise the waterline for the last completed unlocking
        lastCompletedIdxs[callbackAddress][callbackSelector] = request.idx;
        // and delete the request
        delete requestMap[_requestMsgHash];

        // invoke callback
        (success,) = callbackAddress.call(abi.encodeWithSelector(callbackSelector, lockId));

        if (success) {
            emit Completed(lockId, _requestMsgHash, signer1, signer2);
        } else {
            emit Failed(lockId, _requestMsgHash, signer1, signer2);
        }
    }
}

/** @notice  Reclaim the storage of a pending request that is uncompletable.
  *
  * @dev  If a pending request shares the callback (address and selector) of
  * a later request has been completed, then the request can no longer
  * be completed. This function will reclaim the contract storage of the
  * pending request.
  *
  * @param  _requestMsgHash  The request message hash of a pending request.
*/
function deleteUncompletableRequest(bytes32 _requestMsgHash) public {
    Request storage request = requestMap[_requestMsgHash];

    uint256 idx = request.idx;

    require(0 < idx && idx < lastCompletedIdxs[request.callbackAddress][request.callbackSelector],
    "there must be a completed latter request with same callback");

    delete requestMap[_requestMsgHash];
}
```

```
    /** @notice  Extend the time lock of a pending request.
      *
      * @dev  Requests made by the primary account receive the default time lock.
      * This function allows the primary account to apply the extended time lock
      * to one its own requests.
      *
      * @param  _requestMsgHash  The request message hash of a pending request.
     */
    function extendRequestTimeLock(bytes32 _requestMsgHash) public onlyPrimary {
        Request storage request = requestMap[_requestMsgHash];

        // reject ‘null’ results from the map lookup
        // this can only be the case if an unknown `_requestMsgHash` is received
        require(request.callbackAddress != address(0), "reject ‘null’ results from the map lookup");

        // `extendRequestTimeLock` must be idempotent
        require(request.extended != true, "`extendRequestTimeLock` must be idempotent");

        // set the `extended` flag; note that this is never unset
        request.extended = true;

        emit TimeLockExtended(request.timestamp + extendedTimeLock, _requestMsgHash);
    }
}

/** @title  A contract to inherit upgradeable custodianship.
  *
  * @notice  A contract that provides re-usable code for upgradeable
  * custodianship. That custodian may be an account or another contract.
  *
  * @dev  This contract is intended to be inherited by any contract
  * requiring a custodian to control some aspect of its functionality.
  * This contract provides the mechanism for that custodianship to be
  * passed from one custodian to the next.
  *
*/
contract CustodianUpgradeable is LockRequestable {
```

```
// TYPES
/// @dev  The struct type for pending custodian changes.
struct CustodianChangeRequest {
    address proposedNew;
}

// MEMBERS
/// @dev  The address of the account or contract that acts as the custodian.
address public custodian;

/// @dev  The map of lock ids to pending custodian changes.
mapping (bytes32 => CustodianChangeRequest) public custodianChangeReqs;

// CONSTRUCTOR
constructor(
    address _custodian
)
  LockRequestable()
  public
{
    custodian = _custodian;
}

// MODIFIERS
modifier onlyCustodian {
    require(msg.sender == custodian, "only custodian");
    _;
}

// PUBLIC FUNCTIONS
// (UPGRADE)

/** @notice  Requests a change of the custodian associated with this contract.
  *
  * @dev  Returns a unique lock id associated with the request.
  * Anyone can call this function, but confirming the request is authorized
```

```
  * by the custodian.
  *
  * @param  _proposedCustodian  The address of the new custodian.
  * @return  lockId  A unique identifier for this request.
  */
function requestCustodianChange(address _proposedCustodian) public returns (bytes32 lockId) {
    require(_proposedCustodian != address(0), "no null value for `_proposedCustodian`");

    lockId = generateLockId();

    custodianChangeReqs[lockId] = CustodianChangeRequest({
        proposedNew: _proposedCustodian
    });

    emit CustodianChangeRequested(lockId, msg.sender, _proposedCustodian);
}

/** @notice  Confirms a pending change of the custodian associated with this contract.
  *
  * @dev  When called by the current custodian with a lock id associated with a
  * pending custodian change, the `address custodian` member will be updated with the
  * requested address.
  *
  * @param  _lockId  The identifier of a pending change request.
  */
function confirmCustodianChange(bytes32 _lockId) public onlyCustodian {
    custodian = getCustodianChangeReq(_lockId);

    delete custodianChangeReqs[_lockId];

    emit CustodianChangeConfirmed(_lockId, custodian);
}

// PRIVATE FUNCTIONS
function getCustodianChangeReq(bytes32 _lockId) private view returns (address _proposedNew) {
    CustodianChangeRequest storage changeRequest = custodianChangeReqs[_lockId];
```

```
        // reject ‘null’ results from the map lookup
        // this can only be the case if an unknown `_lockId` is received
        require(changeRequest.proposedNew != address(0), "reject ‘null’ results from the map lookup");

        return changeRequest.proposedNew;
    }

    //EVENTS
    /// @dev  Emitted by successful `requestCustodianChange` calls.
    event CustodianChangeRequested(
        bytes32 _lockId,
        address _msgSender,
        address _proposedCustodian
    );

    /// @dev Emitted by successful `confirmCustodianChange` calls.
    event CustodianChangeConfirmed(bytes32 _lockId, address _newCustodian);
}

/** @title  A contract to inherit upgradeable token implementations.
  *
  * @notice  A contract that provides re-usable code for upgradeable
  * token implementations. It itself inherits from `CustodianUpgradable`
  * as the upgrade process is controlled by the custodian.
  *
  * @dev  This contract is intended to be inherited by any contract
  * requiring a reference to the active token implementation, either
  * to delegate calls to it, or authorize calls from it. This contract
  * provides the mechanism for that implementation to be replaced,
  * which constitutes an implementation upgrade.
  *
  */
contract ERC20ImplUpgradeable is CustodianUpgradeable  {

    // TYPES
    /// @dev  The struct type for pending implementation changes.
    struct ImplChangeRequest {
```

```
    address proposedNew;
}

// MEMBERS
// @dev  The reference to the active token implementation.
ERC20Impl public erc20Impl;

/// @dev  The map of lock ids to pending implementation changes.
mapping (bytes32 => ImplChangeRequest) public implChangeReqs;

// CONSTRUCTOR
constructor(address _custodian) CustodianUpgradeable(_custodian) public {
    erc20Impl = ERC20Impl(0x0);
}

// MODIFIERS
modifier onlyImpl {
    require(msg.sender == address(erc20Impl), "only ERC20Impl");
    _;
}

// PUBLIC FUNCTIONS
// (UPGRADE)
/** @notice  Requests a change of the active implementation associated
  * with this contract.
  *
  * @dev  Returns a unique lock id associated with the request.
  * Anyone can call this function, but confirming the request is authorized
  * by the custodian.
  *
  * @param  _proposedImpl  The address of the new active implementation.
  * @return  lockId  A unique identifier for this request.
  */
function requestImplChange(address _proposedImpl) public returns (bytes32 lockId) {
    require(_proposedImpl != address(0), "no null value for `_proposedImpl`");

    lockId = generateLockId();
```

```
    implChangeReqs[lockId] = ImplChangeRequest({
        proposedNew: _proposedImpl
    });

    emit ImplChangeRequested(lockId, msg.sender, _proposedImpl);
}

/** @notice  Confirms a pending change of the active implementation
  * associated with this contract.
  *
  * @dev  When called by the custodian with a lock id associated with a
  * pending change, the `ERC20Impl erc20Impl` member will be updated
  * with the requested address.
  *
  * @param  _lockId  The identifier of a pending change request.
  */
function confirmImplChange(bytes32 _lockId) public onlyCustodian {
    erc20Impl = getImplChangeReq(_lockId);

    delete implChangeReqs[_lockId];

    emit ImplChangeConfirmed(_lockId, address(erc20Impl));
}

// PRIVATE FUNCTIONS
function getImplChangeReq(bytes32 _lockId) private view returns (ERC20Impl _proposedNew) {
    ImplChangeRequest storage changeRequest = implChangeReqs[_lockId];

    // reject ‘null’ results from the map lookup
    // this can only be the case if an unknown `_lockId` is received
    require(changeRequest.proposedNew != address(0), "reject ‘null’ results from the map lookup");

    return ERC20Impl(changeRequest.proposedNew);
}

//EVENTS
```

```
    /// @dev  Emitted by successful `requestImplChange` calls.
    event ImplChangeRequested(
        bytes32 _lockId,
        address _msgSender,
        address _proposedImpl
    );

    /// @dev Emitted by successful `confirmImplChange` calls.
    event ImplChangeConfirmed(bytes32 _lockId, address _newImpl);
}

/** @title  Public interface to ERC20 compliant token.
  *
  * @notice  This contract is a permanent entry point to an ERC20 compliant
  * system of contracts.
  *
  * @dev  This contract contains no business logic and instead
  * delegates to an instance of ERC20Impl. This contract also has no storage
  * that constitutes the operational state of the token. This contract is
  * upgradeable in the sense that the `custodian` can update the
  * `erc20Impl` address, thus redirecting the delegation of business logic.
  * The `custodian` is also authorized to pass custodianship.
  *
*/
contract ERC20Proxy is ERC20Interface, ERC20ImplUpgradeable {

    // MEMBERS
    /// @notice  Returns the name of the token.
    string public name;

    /// @notice  Returns the symbol of the token.
    string public symbol;

    /// @notice  Returns the number of decimals the token uses.
    uint8 public decimals;

    // CONSTRUCTOR
```

```
constructor(
    string memory _name,
    string memory _symbol,
    uint8 _decimals,
    address _custodian
)
    ERC20ImplUpgradeable(_custodian)
    public
{
    name = _name;
    symbol = _symbol;
    decimals = _decimals;
}

// PUBLIC FUNCTIONS
// (ERC20Interface)
/** @notice  Returns the total token supply.
  *
  * @return  the total token supply.
  */
function totalSupply() public view returns (uint256) {
    return erc20Impl.totalSupply();
}

/** @notice  Returns the account balance of another account with an address
  * `_owner`.
  *
  * @return  balance  the balance of account with address `_owner`.
  */
function balanceOf(address _owner) public view returns (uint256 balance) {
    return erc20Impl.balanceOf(_owner);
}
/** @dev Internal use only.
  */
function emitTransfer(address _from, address _to, uint256 _value) public onlyImpl {
    emit Transfer(_from, _to, _value);
}
```

```
/** @notice  Transfers `_value` amount of tokens to address `_to`.
  *
  * @dev Will fire the `Transfer` event. Will revert if the `_from`
  * account balance does not have enough tokens to spend.
  *
  * @return  success  true if transfer completes.
  */
function transfer(address _to, uint256 _value) public returns (bool success) {
    return erc20Impl.transferWithSender(msg.sender, _to, _value);
}

/** @notice  Transfers `_value` amount of tokens from address `_from`
  * to address `_to`.
  *
  * @dev  Will fire the `Transfer` event. Will revert unless the `_from`
  * account has deliberately authorized the sender of the message
  * via some mechanism.
  *
  * @return  success  true if transfer completes.
  */
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
    return erc20Impl.transferFromWithSender(msg.sender, _from, _to, _value);
}

/** @dev Internal use only.
  */
function emitApproval(address _owner, address _spender, uint256 _value) public onlyImpl {
    emit Approval(_owner, _spender, _value);
}

/** @notice  Allows `_spender` to withdraw from your account multiple times,
  * up to the `_value` amount. If this function is called again it
  * overwrites the current allowance with _value.
  *
  * @dev  Will fire the `Approval` event.
  *
```

```
  * @return  success  true if approval completes.
  */
function approve(address _spender, uint256 _value) public returns (bool success) {
    return erc20Impl.approveWithSender(msg.sender, _spender, _value);
}

/** @notice Increases the amount `_spender` is allowed to withdraw from
  * your account.
  * This function is implemented to avoid the race condition in standard
  * ERC20 contracts surrounding the `approve` method.
  *
  * @dev  Will fire the `Approval` event. This function should be used instead of
  * `approve`.
  *
  * @return  success  true if approval completes.
  */
function increaseApproval(address _spender, uint256 _addedValue) public returns (bool success) {
    return erc20Impl.increaseApprovalWithSender(msg.sender, _spender, _addedValue);
}

/** @notice  Decreases the amount `_spender` is allowed to withdraw from
  * your account. This function is implemented to avoid the race
  * condition in standard ERC20 contracts surrounding the `approve` method.
  *
  * @dev  Will fire the `Approval` event. This function should be used
  * instead of `approve`.
  *
  * @return  success  true if approval completes.
  */
function decreaseApproval(address _spender, uint256 _subtractedValue) public returns (bool success) {
    return erc20Impl.decreaseApprovalWithSender(msg.sender, _spender, _subtractedValue);
}

/** @notice  Returns how much `_spender` is currently allowed to spend from
  * `_owner`'s balance.
  *
  * @return  remaining  the remaining allowance.
```

```
     */
    function allowance(address _owner, address _spender) public view returns (uint256 remaining) {
        return erc20Impl.allowance(_owner, _spender);
    }
}

/** @title  ERC20 compliant token balance store.
  *
  * @notice  This contract serves as the store of balances, allowances, and
  * supply for the ERC20 compliant token. No business logic exists here.
  *
  * @dev  This contract contains no business logic and instead
  * is the final destination for any change in balances, allowances, or token
  * supply. This contract is upgradeable in the sense that its custodian can
  * update the `erc20Impl` address, thus redirecting the source of logic that
  * determines how the balances will be updated.
  *
  */
contract ERC20Store is ERC20ImplUpgradeable {

    // MEMBERS
    /// @dev  The total token supply.
    uint256 public totalSupply;

    /// @dev  The mapping of balances.
    mapping (address => uint256) public balances;

    /// @dev  The mapping of allowances.
    mapping (address => mapping (address => uint256)) public allowed;

    // CONSTRUCTOR
    constructor(address _custodian) ERC20ImplUpgradeable(_custodian) public {
        totalSupply = 0;
    }

    // PUBLIC FUNCTIONS
    // (ERC20 Ledger)
```

```
/** @notice  The function to set the total supply of tokens.
  *
  * @dev  Intended for use by token implementation functions
  * that update the total supply. The only authorized caller
  * is the active implementation.
  *
  * @param _newTotalSupply the value to set as the new total supply
  */
function setTotalSupply(
    uint256 _newTotalSupply
)
    public
    onlyImpl
{
    totalSupply = _newTotalSupply;
}

/** @notice  Sets how much `_owner` allows `_spender` to transfer on behalf
  * of `_owner`.
  *
  * @dev  Intended for use by token implementation functions
  * that update spending allowances. The only authorized caller
  * is the active implementation.
  *
  * @param  _owner  The account that will allow an on-behalf-of spend.
  * @param  _spender  The account that will spend on behalf of the owner.
  * @param  _value  The limit of what can be spent.
  */
function setAllowance(
    address _owner,
    address _spender,
    uint256 _value
)
    public
    onlyImpl
{
```

```
    allowed[_owner][_spender] = _value;
}

/** @notice  Sets the balance of `_owner` to `_newBalance`.
  *
  * @dev  Intended for use by token implementation functions
  * that update balances. The only authorized caller
  * is the active implementation.
  *
  * @param  _owner  The account that will hold a new balance.
  * @param  _newBalance  The balance to set.
  */
function setBalance(
    address _owner,
    uint256 _newBalance
)
    public
    onlyImpl
{
    balances[_owner] = _newBalance;
}

/** @notice Adds `_balanceIncrease` to `_owner`'s balance.
  *
  * @dev  Intended for use by token implementation functions
  * that update balances. The only authorized caller
  * is the active implementation.
  * WARNING: the caller is responsible for preventing overflow.
  *
  * @param  _owner  The account that will hold a new balance.
  * @param  _balanceIncrease  The balance to add.
  */
function addBalance(
    address _owner,
    uint256 _balanceIncrease
)
    public
```

```
        onlyImpl
    {
        balances[_owner] = balances[_owner] + _balanceIncrease;
    }
}

/** @title  ERC20 compliant token intermediary contract holding core logic.
  *
  * @notice  This contract serves as an intermediary between the exposed ERC20
  * interface in ERC20Proxy and the store of balances in ERC20Store. This
  * contract contains core logic that the proxy can delegate to
  * and that the store is called by.
  *
  * @dev  This contract contains the core logic to implement the
  * ERC20 specification as well as several extensions.
  * 1. Changes to the token supply.
  * 2. Batched transfers.
  * 3. Relative changes to spending approvals.
  * 4. Delegated transfer control ('sweeping').
  *
  */
contract ERC20Impl is CustodianUpgradeable {

    // TYPES
    /// @dev  The struct type for pending increases to the token supply (print).
    struct PendingPrint {
        address receiver;
        uint256 value;
    }

    // MEMBERS
    /// @dev  The reference to the proxy.
    ERC20Proxy public erc20Proxy;

    /// @dev  The reference to the store.
    ERC20Store public erc20Store;
```

```
/// @dev  The sole authorized caller of delegated transfer control ('sweeping').
address public sweeper;

/** @dev  The static message to be signed by an external account that
  * signifies their permission to forward their balance to any arbitrary
  * address. This is used to consolidate the control of all accounts
  * backed by a shared keychain into the control of a single key.
  * Initialized as the concatenation of the address of this contract
  * and the word "sweep". This concatenation is done to prevent a replay
  * attack in a subsequent contract, where the sweeping message could
  * potentially be replayed to re-enable sweeping ability.
  */
bytes32 public sweepMsg;

/** @dev  The mapping that stores whether the address in question has
  * enabled sweeping its contents to another account or not.
  * If an address maps to "true", it has already enabled sweeping,
  * and thus does not need to re-sign the `sweepMsg` to enact the sweep.
  */
mapping (address => bool) public sweptSet;

/// @dev  The map of lock ids to pending token increases.
mapping (bytes32 => PendingPrint) public pendingPrintMap;

/// @dev The map of blocked addresses.
mapping (address => bool) public blocked;

// CONSTRUCTOR
constructor(
      address _erc20Proxy,
      address _erc20Store,
      address _custodian,
      address _sweeper
)
    CustodianUpgradeable(_custodian)
    public
{
```

```
    require(_sweeper != address(0), "no null value for `_sweeper`");
    erc20Proxy = ERC20Proxy(_erc20Proxy);
    erc20Store = ERC20Store(_erc20Store);

    sweeper = _sweeper;
    sweepMsg = keccak256(abi.encodePacked(address(this), "sweep"));
}

// MODIFIERS
modifier onlyProxy {
    require(msg.sender == address(erc20Proxy), "only ERC20Proxy");
    _;
}
modifier onlySweeper {
    require(msg.sender == sweeper, "only sweeper");
    _;
}



/** @notice  Core logic of the ERC20 `approve` function.
  *
  * @dev  This function can only be called by the referenced proxy,
  * which has an `approve` function.
  * Every argument passed to that function as well as the original
  * `msg.sender` gets passed to this function.
  * NOTE: approvals for the zero address (unspendable) are disallowed.
  *
  * @param  _sender  The address initiating the approval in a proxy.
  */
function approveWithSender(
    address _sender,
    address _spender,
    uint256 _value
)
    public
    onlyProxy
    returns (bool success)
```

```
{
    require(_spender != address(0), "no null value for `_spender`");
    require(blocked[_sender] != true, "_sender must not be blocked");
    require(blocked[_spender] != true, "_spender must not be blocked");
    erc20Store.setAllowance(_sender, _spender, _value);
    erc20Proxy.emitApproval(_sender, _spender, _value);
    return true;
}

/** @notice  Core logic of the `increaseApproval` function.
  *
  * @dev  This function can only be called by the referenced proxy,
  * which has an `increaseApproval` function.
  * Every argument passed to that function as well as the original
  * `msg.sender` gets passed to this function.
  * NOTE: approvals for the zero address (unspendable) are disallowed.
  *
  * @param  _sender  The address initiating the approval.
  */
function increaseApprovalWithSender(
    address _sender,
    address _spender,
    uint256 _addedValue
)
    public
    onlyProxy
    returns (bool success)
{
    require(_spender != address(0),"no null value for_spender");
    require(blocked[_sender] != true, "_sender must not be blocked");
    require(blocked[_spender] != true, "_spender must not be blocked");
    uint256 currentAllowance = erc20Store.allowed(_sender, _spender);
    uint256 newAllowance = currentAllowance + _addedValue;

    require(newAllowance >= currentAllowance, "new allowance must not be smaller than previous");

    erc20Store.setAllowance(_sender, _spender, newAllowance);
```

```
    erc20Proxy.emitApproval(_sender, _spender, newAllowance);
    return true;
}

/** @notice  Core logic of the `decreaseApproval` function.
  *
  * @dev  This function can only be called by the referenced proxy,
  * which has a `decreaseApproval` function.
  * Every argument passed to that function as well as the original
  * `msg.sender` gets passed to this function.
  * NOTE: approvals for the zero address (unspendable) are disallowed.
  *
  * @param  _sender  The address initiating the approval.
  */
function decreaseApprovalWithSender(
    address _sender,
    address _spender,
    uint256 _subtractedValue
)
    public
    onlyProxy
    returns (bool success)
{
    require(_spender != address(0), "no unspendable approvals"); // disallow unspendable approvals
    require(blocked[_sender] != true, "_sender must not be blocked");
    require(blocked[_spender] != true, "_spender must not be blocked");
    uint256 currentAllowance = erc20Store.allowed(_sender, _spender);
    uint256 newAllowance = currentAllowance - _subtractedValue;

    require(newAllowance <= currentAllowance, "new allowance must not be smaller than previous");

    erc20Store.setAllowance(_sender, _spender, newAllowance);
    erc20Proxy.emitApproval(_sender, _spender, newAllowance);
    return true;
}

/** @notice  Requests an increase in the token supply, with the newly created
```

```
  * tokens to be added to the balance of the specified account.
  *
  * @dev  Returns a unique lock id associated with the request.
  * Anyone can call this function, but confirming the request is authorized
  * by the custodian.
  * NOTE: printing to the zero address is disallowed.
  *
  * @param  _receiver  The receiving address of the print, if confirmed.
  * @param  _value  The number of tokens to add to the total supply and the
  * balance of the receiving address, if confirmed.
  *
  * @return  lockId  A unique identifier for this request.
  */
function requestPrint(address _receiver, uint256 _value) public returns (bytes32 lockId) {
    require(_receiver != address(0), "no null value for `_receiver`");
    require(blocked[msg.sender] != true, "account blocked");
    require(blocked[_receiver] != true, "_receiver must not be blocked");
    lockId = generateLockId();

    pendingPrintMap[lockId] = PendingPrint({
        receiver: _receiver,
        value: _value
    });

    emit PrintingLocked(lockId, _receiver, _value);
}

/** @notice  Confirms a pending increase in the token supply.
  *
  * @dev  When called by the custodian with a lock id associated with a
  * pending increase, the amount requested to be printed in the print request
  * is printed to the receiving address specified in that same request.
  * NOTE: this function will not execute any print that would overflow the
  * total supply, but it will not revert either.
  *
  * @param  _lockId  The identifier of a pending print request.
  */
```

```
function confirmPrint(bytes32 _lockId) public onlyCustodian {
    PendingPrint storage print = pendingPrintMap[_lockId];

    // reject ‘null’ results from the map lookup
    // this can only be the case if an unknown `_lockId` is received
    address receiver = print.receiver;
    require (receiver != address(0), "unknown `_lockId`");
    uint256 value = print.value;

    delete pendingPrintMap[_lockId];

    uint256 supply = erc20Store.totalSupply();
    uint256 newSupply = supply + value;
    if (newSupply >= supply) {
      erc20Store.setTotalSupply(newSupply);
      erc20Store.addBalance(receiver, value);

      emit PrintingConfirmed(_lockId, receiver, value);
      erc20Proxy.emitTransfer(address(0), receiver, value);
    }
}

/** @notice  Burns the specified value from the sender's balance.
  *
  * @dev  Sender's balanced is subtracted by the amount they wish to burn.
  *
  * @param  _value  The amount to burn.
  *
  * @return success true if the burn succeeded.
  */
function burn(uint256 _value) public returns (bool success) {
    require(blocked[msg.sender] != true, "account blocked");
    uint256 balanceOfSender = erc20Store.balances(msg.sender);
    require(_value <= balanceOfSender, "disallow burning more, than amount of the balance");

    erc20Store.setBalance(msg.sender, balanceOfSender - _value);
    erc20Store.setTotalSupply(erc20Store.totalSupply() - _value);
```

```
    erc20Proxy.emitTransfer(msg.sender, address(0), _value);

    return true;
}

 /** @notice  Burns the specified value from the balance in question.
  *
  * @dev  Suspected balance is subtracted by the amount which will be burnt.
  *
  * @dev If the suspected balance has less than the amount requested, it will be set to 0.
  *
  * @param  _from  The address of suspected balance.
  *
  * @param  _value  The amount to burn.
  *
  * @return success true if the burn succeeded.
  */
function burn(address _from, uint256 _value) public onlyCustodian returns (bool success) {
    uint256 balance = erc20Store.balances(_from);
    if(_value <= balance){
        erc20Store.setBalance(_from, balance - _value);
        erc20Store.setTotalSupply(erc20Store.totalSupply() - _value);
        erc20Proxy.emitTransfer(_from, address(0), _value);
        emit Wiped(_from, _value, _value, balance - _value);
    }
    else {
        erc20Store.setBalance(_from,0);
        erc20Store.setTotalSupply(erc20Store.totalSupply() - balance);
        erc20Proxy.emitTransfer(_from, address(0), balance);
        emit Wiped(_from, _value, balance, 0);
    }
    return true;
}

/** @notice  A function for a sender to issue multiple transfers to multiple
  * different addresses at once. This function is implemented for gas
```

```
  * considerations when someone wishes to transfer, as one transaction is
  * cheaper than issuing several distinct individual `transfer` transactions.
  *
  * @dev  By specifying a set of destination addresses and values, the
  * sender can issue one transaction to transfer multiple amounts to
  * distinct addresses, rather than issuing each as a separate
  * transaction. The `_tos` and `_values` arrays must be equal length, and
  * an index in one array corresponds to the same index in the other array
  * (e.g. `_tos[0]` will receive `_values[0]`, `_tos[1]` will receive
  * `_values[1]`, and so on.)
  * NOTE: transfers to the zero address are disallowed.
  *
  * @param  _tos  The destination addresses to receive the transfers.
  * @param  _values  The values for each destination address.
  * @return  success  If transfers succeeded.
  */
function batchTransfer(address[] memory _tos, uint256[] memory _values) public returns (bool success) {
    require(_tos.length == _values.length, "_tos and _values must be the same length");
    require(blocked[msg.sender] != true, "account blocked");
    uint256 numTransfers = _tos.length;
    uint256 senderBalance = erc20Store.balances(msg.sender);

    for (uint256 i = 0; i < numTransfers; i++) {
      address to = _tos[i];
      require(to != address(0), "no null values for _tos");
      require(blocked[to] != true, "_tos must not be blocked");
      uint256 v = _values[i];
      require(senderBalance >= v, "insufficient funds");

      if (msg.sender != to) {
        senderBalance -= v;
        erc20Store.addBalance(to, v);
      }
      erc20Proxy.emitTransfer(msg.sender, to, v);
    }

    erc20Store.setBalance(msg.sender, senderBalance);
```

```
        return true;
    }

    /** @notice  Enables the delegation of transfer control for many
      * accounts to the sweeper account, transferring any balances
      * as well to the given destination.
      *
      * @dev  An account delegates transfer control by signing the
      * value of `sweepMsg`. The sweeper account is the only authorized
      * caller of this function, so it must relay signatures on behalf
      * of accounts that delegate transfer control to it. Enabling
      * delegation is idempotent and permanent. If the account has a
      * balance at the time of enabling delegation, its balance is
      * also transferred to the given destination account `_to`.
      * NOTE: transfers to the zero address are disallowed.
      *
      * @param  _vs  The array of recovery byte components of the ECDSA signatures.
      * @param  _rs  The array of 'R' components of the ECDSA signatures.
      * @param  _ss  The array of 'S' components of the ECDSA signatures.
      * @param  _to  The destination for swept balances.
      */
    function enableSweep(uint8[] memory _vs, bytes32[] memory _rs, bytes32[] memory _ss, address _to) public onlySweeper {
        require(_to != address(0), "no null value for `_to`");
        require(blocked[_to] != true, "_to must not be blocked");
        require((_vs.length == _rs.length) && (_vs.length == _ss.length), "_vs[], _rs[], _ss lengths are different");

        uint256 numSignatures = _vs.length;
        uint256 sweptBalance = 0;

        for (uint256 i = 0; i < numSignatures; ++i) {
            address from = ecrecover(keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32",sweepMsg)), _vs[i],
_rs[i], _ss[i]);
            require(blocked[from] != true, "_froms must not be blocked");
            // ecrecover returns 0 on malformed input
            if (from != address(0)) {
                sweptSet[from] = true;
```

```
            uint256 fromBalance = erc20Store.balances(from);

            if (fromBalance > 0) {
                sweptBalance += fromBalance;

                erc20Store.setBalance(from, 0);

                erc20Proxy.emitTransfer(from, _to, fromBalance);
            }
        }
    }

    if (sweptBalance > 0) {
      erc20Store.addBalance(_to, sweptBalance);
    }
}

/** @notice  For accounts that have delegated, transfer control
  * to the sweeper, this function transfers their balances to the given
  * destination.
  *
  * @dev The sweeper account is the only authorized caller of
  * this function. This function accepts an array of addresses to have their
  * balances transferred for gas efficiency purposes.
  * NOTE: any address for an account that has not been previously enabled
  * will be ignored.
  * NOTE: transfers to the zero address are disallowed.
  *
  * @param  _froms  The addresses to have their balances swept.
  * @param  _to  The destination address of all these transfers.
  */
function replaySweep(address[] memory _froms, address _to) public onlySweeper {
    require(_to != address(0), "no null value for `_to`");
    require(blocked[_to] != true, "_to must not be blocked");
    uint256 lenFroms = _froms.length;
    uint256 sweptBalance = 0;
```

```
    for (uint256 i = 0; i < lenFroms; ++i) {
        address from = _froms[i];
        require(blocked[from] != true, "_froms must not be blocked");
        if (sweptSet[from]) {
            uint256 fromBalance = erc20Store.balances(from);

            if (fromBalance > 0) {
                sweptBalance += fromBalance;

                erc20Store.setBalance(from, 0);

                erc20Proxy.emitTransfer(from, _to, fromBalance);
            }
        }
    }

    if (sweptBalance > 0) {
        erc20Store.addBalance(_to, sweptBalance);
    }
}

/** @notice  Core logic of the ERC20 `transferFrom` function.
  *
  * @dev  This function can only be called by the referenced proxy,
  * which has a `transferFrom` function.
  * Every argument passed to that function as well as the original
  * `msg.sender` gets passed to this function.
  * NOTE: transfers to the zero address are disallowed.
  *
  * @param  _sender  The address initiating the transfer in a proxy.
  */
function transferFromWithSender(
    address _sender,
    address _from,
    address _to,
    uint256 _value
```

```
)
    public
    onlyProxy
    returns (bool success)
{
    require(_to != address(0), "no null values for `_to`");
    require(blocked[_sender] != true, "_sender must not be blocked");
    require(blocked[_from] != true, "_from must not be blocked");
    require(blocked[_to] != true, "_to must not be blocked");

    uint256 balanceOfFrom = erc20Store.balances(_from);
    require(_value <= balanceOfFrom, "insufficient funds on `_from` balance");

    uint256 senderAllowance = erc20Store.allowed(_from, _sender);
    require(_value <= senderAllowance, "insufficient allowance amount");

    erc20Store.setBalance(_from, balanceOfFrom - _value);
    erc20Store.addBalance(_to, _value);

    erc20Store.setAllowance(_from, _sender, senderAllowance - _value);

    erc20Proxy.emitTransfer(_from, _to, _value);

    return true;
}

/** @notice  Core logic of the ERC20 `transfer` function.
  *
  * @dev  This function can only be called by the referenced proxy,
  * which has a `transfer` function.
  * Every argument passed to that function as well as the original
  * `msg.sender` gets passed to this function.
  * NOTE: transfers to the zero address are disallowed.
  *
  * @param  _sender  The address initiating the transfer in a proxy.
  */
function transferWithSender(
```

```
    address _sender,
    address _to,
    uint256 _value
)
    public
    onlyProxy
    returns (bool success)
{
    require(_to != address(0), "no null value for `_to`");
    require(blocked[_sender] != true, "_sender must not be blocked");
    require(blocked[_to] != true, "_to must not be blocked");

    uint256 balanceOfSender = erc20Store.balances(_sender);
    require(_value <= balanceOfSender, "insufficient funds");

    erc20Store.setBalance(_sender, balanceOfSender - _value);
    erc20Store.addBalance(_to, _value);

    erc20Proxy.emitTransfer(_sender, _to, _value);

    return true;
}

/** @notice  Transfers the specified value from the balance in question.
  *
  * @dev  Suspected balance is subtracted by the amount which will be transferred.
  *
  * @dev If the suspected balance has less than the amount requested, it will be set to 0.
  *
  * @param  _from  The address of suspected balance.
  *
  * @param  _value  The amount to transfer.
  *
  * @return success true if the transfer succeeded.
  */
function forceTransfer(
    address _from,
```

```
    address _to,
    uint256 _value
)
    public
    onlyCustodian
    returns (bool success)
{
    require(_to != address(0), "no null value for `_to`");
    uint256 balanceOfSender = erc20Store.balances(_from);
    if(_value <= balanceOfSender) {
        erc20Store.setBalance(_from, balanceOfSender - _value);
        erc20Store.addBalance(_to, _value);

        erc20Proxy.emitTransfer(_from, _to, _value);
    } else {
        erc20Store.setBalance(_from, 0);
        erc20Store.addBalance(_to, balanceOfSender);

        erc20Proxy.emitTransfer(_from, _to, balanceOfSender);
    }

    return true;
}

// METHODS (ERC20 sub interface impl.)
/// @notice  Core logic of the ERC20 `totalSupply` function.
function totalSupply() public view returns (uint256) {
    return erc20Store.totalSupply();
}

/// @notice  Core logic of the ERC20 `balanceOf` function.
function balanceOf(address _owner) public view returns (uint256 balance) {
    return erc20Store.balances(_owner);
}

/// @notice  Core logic of the ERC20 `allowance` function.
function allowance(address _owner, address _spender) public view returns (uint256 remaining) {
```

```
        return erc20Store.allowed(_owner, _spender);
    }

    /// @dev internal use only.
    function blockWallet(address wallet) public onlyCustodian returns (bool success) {
        blocked[wallet] = true;
        return true;
    }

    /// @dev internal use only.
    function unblockWallet(address wallet) public onlyCustodian returns (bool success) {
        blocked[wallet] = false;
        return true;
    }

    // EVENTS
    /// @dev  Emitted by successful `requestPrint` calls.
    event PrintingLocked(bytes32 _lockId, address _receiver, uint256 _value);

    /// @dev Emitted by successful `confirmPrint` calls.
    event PrintingConfirmed(bytes32 _lockId, address _receiver, uint256 _value);

    /** @dev Emitted by successful `confirmWipe` calls.
      *
      * @param _value Amount requested to be burned.
      *
      * @param _burned Amount which was burned.
      *
      * @param _balance Amount left on account after burn.
      *
      * @param _from Account which balance was burned.
      */
     event Wiped(address _from, uint256 _value, uint256 _burned, uint _balance);
}

/** @title  A contact to govern hybrid control over increases to the token supply and managing accounts.
  *
```

```
 * @notice  A contract that acts as a custodian of the active token
 * implementation, and an intermediary between it and the ‘true’ custodian.
 * It preserves the functionality of direct custodianship as well as granting
 * limited control of token supply increases to an additional key.
 *
 * @dev  This contract is a layer of indirection between an instance of
 * ERC20Impl and a custodian. The functionality of the custodianship over
 * the token implementation is preserved (printing and custodian changes),
 * but this contract adds the ability for an additional key
 * (the 'controller') to increase the token supply up to a ceiling,
 * and this supply ceiling can only be raised by the custodian.
 *
 */
contract Controller is LockRequestable {

    // TYPES
    /// @dev The struct type for pending ceiling raises.
    struct PendingCeilingRaise {
        uint256 raiseBy;
    }

    /// @dev The struct type for pending wipes.
    struct wipeAddress {
        uint256 value;
        address from;
    }

    /// @dev The struct type for pending force transfer requests.
    struct forceTransferRequest {
        uint256 value;
        address from;
        address to;
    }

    // MEMBERS
    /// @dev  The reference to the active token implementation.
    ERC20Impl public erc20Impl;
```

```
/// @dev  The address of the account or contract that acts as the custodian.
Custodian public custodian;

/** @dev  The sole authorized caller of limited printing.
  * This account is also authorized to lower the supply ceiling and
  * wiping suspected accounts or force transferring funds from them.
  */
address public controller;

/** @dev  The maximum that the token supply can be increased to
  * through the use of the limited printing feature.
  * The difference between the current total supply and the supply
  * ceiling is what is available to the 'controller' account.
  * The value of the ceiling can only be increased by the custodian.
  */
uint256 public totalSupplyCeiling;

/// @dev  The map of lock ids to pending ceiling raises.
mapping (bytes32 => PendingCeilingRaise) public pendingRaiseMap;

/// @dev  The map of lock ids to pending wipes.
mapping (bytes32 => wipeAddress[]) public pendingWipeMap;

/// @dev  The map of lock ids to pending force transfer requests.
mapping (bytes32 => forceTransferRequest) public pendingForceTransferRequestMap;

// CONSTRUCTOR
constructor(
    address _erc20Impl,
    address _custodian,
    address _controller,
    uint256 _initialCeiling
)
    public
{
    erc20Impl = ERC20Impl(_erc20Impl);
```

```
    custodian = Custodian(_custodian);
    controller = _controller;
    totalSupplyCeiling = _initialCeiling;
}

// MODIFIERS
modifier onlyCustodian {
    require(msg.sender == address(custodian), "only custodian");
    _;
}
modifier onlyController {
    require(msg.sender == controller, "only controller");
    _;
}

modifier onlySigner {
    require(custodian.signerSet(msg.sender) == true, "only signer");
    _;
}

/** @notice  Increases the token supply, with the newly created tokens
  * being added to the balance of the specified account.
  *
  * @dev  The function checks that the value to print does not
  * exceed the supply ceiling when added to the current total supply.
  * NOTE: printing to the zero address is disallowed.
  *
  * @param  _receiver  The receiving address of the print.
  * @param  _value  The number of tokens to add to the total supply and the
  * balance of the receiving address.
  */
function limitedPrint(address _receiver, uint256 _value) public onlyController {
    uint256 totalSupply = erc20Impl.totalSupply();
    uint256 newTotalSupply = totalSupply + _value;

    require(newTotalSupply >= totalSupply, "new total supply overflow");
    require(newTotalSupply <= totalSupplyCeiling, "total supply ceiling overflow");
```

```
    erc20Impl.confirmPrint(erc20Impl.requestPrint(_receiver, _value));
}

/** @notice  Requests wipe of suspected accounts.
  *
  * @dev  Returns a unique lock id associated with the request.
  * Only controller can call this function, and only the custodian
  * can confirm the request.
  *
  * @param  _froms  The array of suspected accounts.
  *
  * @param  _values  array of amounts by which suspected accounts will be wiped.
  *
  * @return  lockId  A unique identifier for this request.
  */
function requestWipe(address[] memory _froms, uint256[] memory _values) public onlyController returns (bytes32 lockId) {
    require(_froms.length == _values.length, "_froms[] and _values[] must be same length");
    lockId = generateLockId();
    uint256 amount = _froms.length;

    for(uint256 i = 0; i < amount; i++) {
        address from = _froms[i];
        uint256 value = _values[i];
        pendingWipeMap[lockId].push(wipeAddress(value, from));
    }

    emit WipeRequested(lockId);

    return lockId;
}

/** @notice  Confirms a pending wipe of suspected accounts.
  *
  * @dev  When called by the custodian with a lock id associated with a
  * pending wipe, the amount requested is burned from the suspected accounts.
  *
  * @param  _lockId  The identifier of a pending wipe request.
```

```
  */
function confirmWipe(bytes32 _lockId) public onlyCustodian {
    uint256 amount = pendingWipeMap[_lockId].length;
    for(uint256 i = 0; i < amount; i++) {
        wipeAddress memory addr = pendingWipeMap[_lockId][i];
        address from = addr.from;
        uint256 value = addr.value;
        erc20Impl.burn(from, value);
    }

    delete pendingWipeMap[_lockId];

    emit WipeCompleted(_lockId);
}

/** @notice  Requests force transfer from the suspected account.
  *
  * @dev  Returns a unique lock id associated with the request.
  * Only controller can call this function, and only the custodian
  * can confirm the request.
  *
  * @param  _from  address of suspected account.
  *
  * @param  _to  address of reciever.
  *
  * @param  _value  amount which will be transferred.
  *
  * @return  lockId  A unique identifier for this request.
  */
function requestForceTransfer(address _from, address _to, uint256 _value) public onlyController returns (bytes32 lockId) {
    lockId = generateLockId();
    require (_value != 0, "no zero value transfers");
    pendingForceTransferRequestMap[lockId] = forceTransferRequest(_value, _from, _to);

    emit ForceTransferRequested(lockId, _from, _to, _value);

    return lockId;
```

```
}

/** @notice  Confirms a pending force transfer request.
  *
  * @dev  When called by the custodian with a lock id associated with a
  * pending transfer request, the amount requested is transferred from the suspected account.
  *
  * @param  _lockId  The identifier of a pending transfer request.
  */
function confirmForceTransfer(bytes32 _lockId) public onlyCustodian {
    address from = pendingForceTransferRequestMap[_lockId].from;
    address to = pendingForceTransferRequestMap[_lockId].to;
    uint256 value = pendingForceTransferRequestMap[_lockId].value;

    delete pendingForceTransferRequestMap[_lockId];

    erc20Impl.forceTransfer(from, to, value);

    emit ForceTransferCompleted(_lockId, from, to, value);
}

/** @notice  Requests an increase to the supply ceiling.
  *
  * @dev  Returns a unique lock id associated with the request.
  * Anyone can call this function, but confirming the request is authorized
  * by the custodian.
  *
  * @param  _raiseBy  The amount by which to raise the ceiling.
  *
  * @return  lockId  A unique identifier for this request.
  */
function requestCeilingRaise(uint256 _raiseBy) public returns (bytes32 lockId) {
    require(_raiseBy != 0, "no zero ceiling raise");

    lockId = generateLockId();

    pendingRaiseMap[lockId] = PendingCeilingRaise({
```

```
        raiseBy: _raiseBy
    });

    emit CeilingRaiseLocked(lockId, _raiseBy);
}

/** @notice  Confirms a pending increase in the token supply.
  *
  * @dev  When called by the custodian with a lock id associated with a
  * pending ceiling increase, the amount requested is added to the
  * current supply ceiling.
  * NOTE: this function will not execute any raise that would overflow the
  * supply ceiling, but it will not revert either.
  *
  * @param  _lockId  The identifier of a pending ceiling raise request.
  */
function confirmCeilingRaise(bytes32 _lockId) public onlyCustodian {
    PendingCeilingRaise storage pendingRaise = pendingRaiseMap[_lockId];

    // copy locals of references to struct members
    uint256 raiseBy = pendingRaise.raiseBy;
    // accounts for a gibberish _lockId
    require(raiseBy != 0, "no gibberish _lockId");

    delete pendingRaiseMap[_lockId];

    uint256 newCeiling = totalSupplyCeiling + raiseBy;
    // overflow check
    if (newCeiling >= totalSupplyCeiling) {
        totalSupplyCeiling = newCeiling;

        emit CeilingRaiseConfirmed(_lockId, raiseBy, newCeiling);
    }
}

/** @notice  Lowers the supply ceiling, further constraining the bound of
  * what can be printed by the controller.
```

```
  *
  * @dev  The controller is the sole authorized caller of this function,
  * so it is the only account that can elect to lower its limit to increase
  * the token supply.
  *
  * @param  _lowerBy  The amount by which to lower the supply ceiling.
  */
function lowerCeiling(uint256 _lowerBy) public onlyController {
    uint256 newCeiling = totalSupplyCeiling - _lowerBy;
    // overflow check
    require(newCeiling <= totalSupplyCeiling, "totalSupplyCeiling overflow");
    totalSupplyCeiling = newCeiling;

    emit CeilingLowered(_lowerBy, newCeiling);
}

/** @notice  Pass-through control of print confirmation, allowing this
  * contract's custodian to act as the custodian of the associated
  * active token implementation.
  *
  * @dev  This contract is the direct custodian of the active token
  * implementation, but this function allows this contract's custodian
  * to act as though it were the direct custodian of the active
  * token implementation. Therefore the custodian retains control of
  * unlimited printing.
  *
  * @param  _lockId  The identifier of a pending print request in
  * the associated active token implementation.
  */
function confirmPrintProxy(bytes32 _lockId) public onlyCustodian {
    erc20Impl.confirmPrint(_lockId);
}

/** @notice  Pass-through control of custodian change confirmation,
  * allowing this contract's custodian to act as the custodian of
  * the associated active token implementation.
  *
```

```
  * @dev  This contract is the direct custodian of the active token
  * implementation, but this function allows this contract's custodian
  * to act as though it were the direct custodian of the active
  * token implementation. Therefore the custodian retains control of
  * custodian changes.
  *
  * @param  _lockId  The identifier of a pending custodian change request
  * in the associated active token implementation.
  */
function confirmCustodianChangeProxy(bytes32 _lockId) public onlyCustodian {
    erc20Impl.confirmCustodianChange(_lockId);
}

/** @notice  Blocks all transactions with a wallet.
  *
  * @dev Only signers from custodian are authorized to call this function
  *
  * @param  wallet account which will be blocked.
  */
function blockWallet(address wallet) public onlySigner {
    erc20Impl.blockWallet(wallet);
    emit Blocked(wallet);
}

/** @notice Unblocks all transactions with a wallet.
  *
  * @dev Only signers from custodian are authorized to call this function
  *
  * @param  wallet account which will be unblocked.
  */
function unblockWallet(address wallet) public onlySigner {
    erc20Impl.unblockWallet(wallet);
    emit Unblocked(wallet);
}

// EVENTS
/// @dev  Emitted by successful `requestCeilingRaise` calls.
```

```
    event CeilingRaiseLocked(bytes32 _lockId, uint256 _raiseBy);

    /// @dev  Emitted by successful `confirmCeilingRaise` calls.
    event CeilingRaiseConfirmed(bytes32 _lockId, uint256 _raiseBy, uint256 _newCeiling);

    /// @dev  Emitted by successful `lowerCeiling` calls.
    event CeilingLowered(uint256 _lowerBy, uint256 _newCeiling);

    /// @dev  Emitted by successful `blockWallet` calls.
    event Blocked(address _wallet);

    /// @dev  Emitted by successful `unblockWallet` calls.
    event Unblocked(address _wallet);

     /// @dev  Emitted by successful `requestForceTransfer` calls.
    event ForceTransferRequested(bytes32 _lockId, address _from, address _to, uint256 _value);

    /// @dev  Emitted by successful `confirmForceTransfer` calls.
    event ForceTransferCompleted(bytes32 _lockId, address _from, address _to, uint256 _value);

    /// @dev  Emitted by successful `requestWipe` calls.
    event WipeRequested(bytes32 _lockId);

    /// @dev  Emitted by successful `confirmWipe` calls.
    event WipeCompleted(bytes32 _lockId);
}
```

kaspersky

www.kaspersky.com/
www.securelist.com