DeCash

Smart Contracts Audit

v2.0

Vittorio Minacori
BC1 - Blockchain Pioneers

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bugfree status. The audit documentation is for discussion purposes only.

Overview

The audit was performed by **BC1 - Blockchain Pioneers** (https://www.bc1.tech) on all the Smart Contracts provided by the **DeCash** team.

This document can be found here blob/v2.0/BC1/docs/DeCash-BC1-Audit-Report-V2.0.pdf

Contracts audited are in the **DeCash** repository at github.com/ConKreter/Decash-SmartContract/tree/v2.0.
The audit is based on the commit fbf949614767e29cbe333a0a3b9987463c865fa5.

Analysis is in the **audit** branch.

By the **DeCash** settings we are assuming that smart contracts will be compiled and deployed using ***solc v0.7.6+commit.7338295f.Emscripten.clang.***
NOTE: also if solc 0.8.0 is released we are considering that smart contracts were written when ^0.7.0 was the latest version so we are considering 0.7.6 as latest.

Brief

Audit was performed on **DeCashToken.sol** which extends **DeCashBase.sol**, **DeCashMultisignature.sol** and **ERC20.sol**.

DeCashToken contract will be used as a "template" to generate other tokens with the same behaviours in the "contracts/currencies" folder.

DeCashToken contract is intended to be upgradeable so it requires **DecashStorage**, **DeCashUpgrade**, **DeCashRole** and **DeCashProxy** to work. The above contracts have been audited too.

Best Practices

Blockchain developers following best practices, make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfillment of general guidelines. Avoiding code duplication is a good example of a good engineering practice which increases the potential of any security audit. We consider a list of few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain.

Hard Requirements

✅ The code is provided as a Git repository to allow the review of future code changes**.**

✅ Code duplication is minimal, or justified and documented.

❌ Libraries are properly referred to as package dependencies, including the specific version(s).

✅ The code compiles with the latest Solidity compiler version.

✅ There are no compiler warnings, or warnings are documented.

❌ There are tests.

❌ The test coverage is available or can be obtained easily.

Soft Requirements

⚠️ The code is well documented.

✅ Functions are grouped and named according either to the Solidity guidelines, or to their functionality.

✅ Use of *modifiers* to avoid code duplication.

✅ Use of *SafeMath* library to prevent integer overflow (unnecessary since solidity 0.8.x).

Audit Details

Arithmetic Safety Audit ✅

The arithmetic security audit is divided into three parts: integer overflow audit, integer underflow audit and operation precision audit.

Integer Overflow Audit ✅

Solidity can handle 256 bits of data at most. When the maximum number increases, it will overflow. If the integer overflow occurs in the transfer logic, it will make the amount of transfer funds miscalculated, resulting in serious capital risk.

Integer Underflow Audit ✅

Solidity can handle 256 bits of data at most. When the minimum number decreases, it will underflow. If the integer underflow occurs in the transfer logic, it will make the amount of transfer funds miscalculated and lead to serious capital risk.

Operation Precision Audit ✅

Solidity performs type coercion in the process of multiplication and division. If the precision risk is included in the operation of capital variable, it will lead to user transfer logic error and capital loss.

Audit Result: Passed
Security Recommendation: No.

Competitive Competition Audit ✅

The competitive competition audit is divided into two parts: reentrancy audit and transaction ordering dependence audit. With competitive vulnerabilities, an attacker can modify the output of a program by adjusting the execution process of transactions with a certain probability.

Reentrancy Audit ✅

Reentrancy occurs when external contract calls are allowed to make new calls to the calling contract before the initial execution is complete. For a function, this means that the contract state may change in the middle of its execution as a result of a call to an untrusted contract or the use of a low level function with an external address.

Transaction Ordering Dependence Audit ✅
Since miners always get rewarded via gas fees for running code on behalf of externally owned addresses (EOA), users can specify higher fees to have their transactions mined more quickly. Since the Ethereum blockchain is public, everyone can see the contents of others' pending transactions. This means if a given user is revealing the solution to a puzzle or other valuable secret, a malicious user can steal the solution and copy their transaction with higher fees to preempt the original solution. If developers of smart contracts are not careful, this situation can lead to practical and devastating front-running attacks.

Audit Result: Passed
Security Recommendation: No.

Access Control Audit ⚠️
Access control audit is divided into two parts: privilege vulnerability audit and overprivileged audit.

Privilege Vulnerability Audit ✅
Smart contracts with privilege vulnerability, attackers can weigh their own accounts to gain higher execution privileges.

Overprivileged Audit ⚠️
Overprivileged auditing focuses on whether there are special user privileges in audit contracts, such as allowing a user to unlimitedly mine tokens.

Audit Result: Warning
Security Recommendation: Check the Audit Results section for details.

Security Design Audit ✅
Security design audit is divided into four parts: compiler version security, hard-coded address security, sensitive function usage security and function return value security.

Compiler Version Security Audit ✅
Compiler version security focuses on whether the smart contract explicitly indicates the compiler version and whether the compiler version used is too low to throw an exception.

Hard Coded Address Security Audit ✅
Hard-coded address security audit static addressed in the smart contract to check whether there is an exception to the external contract, thus affecting the execution of this contract.

Sensitive Functions Audit ✅
Sensitive functions audit checks whether the smart contract uses the not recommended functions such as fallback, call and tx.origin.

Function Return Value Audit ✅
Function return value audit mainly analyzes whether the function correctly throws an exception, correctly returns to the state of the transaction.

Audit Result: Passed
Security Recommendation: No.

Denial of Service Audit ✅

Denial of service attack sometimes can put the smart contract offline forever by maliciously behaving when being the recipient of a transaction, artificially increasing the gas necessary to compute a function, abusing access controls to access private components of smart contracts, taking advantage of mixups and negligence and so on.

Audit Result: Passed
Security Recommendation: No.

Gas Optimization Audit ✅

If the computation of a function in a smart contract is too complex, such as the batch transfer to a variable-length array through a loop, it is very easy to cause the gas fee beyond the block's gas Limit resulting in transaction execution failure.

Audit Result: Passed
Security Recommendation: Check the Audit Results section for details.

Design Logic Audit ✅

In addition to vulnerabilities, there are logic problems in the process of code implementation, resulting in abnormal execution results.

Audit Result: Passed
Security Recommendation: Check the Audit Results section for details.

Audit Results

Low severity (SOLVED) ⚠️

- **Overprivileged Audit:**
    - **DeCashToken** has a lot of methods controlled by the "whenNotPaused" modifier. This means that super users (owner or admins) can decide to pause tokens and lock users balances. This will be better analyzed below.
    We initially marked this as a low severity issue and asked DeCash to explain behaviours.
    We removed this severity. See explanation in Considerations section.

    - **DeCashToken** has no way to define the maximum number of tokens mintable. If there is a maximum number of tokens to be ever minted we would suggest to set a max cap of mintable tokens or add a *finishMinting* function to disable minting. Since there is no finish minting method, minters could generate more tokens than declared.
    We initially marked this as a low severity issue and asked DeCash to explain behaviours.
    We removed this severity. See explanation in Considerations section.

Note: *approve()* is affected by the issue related on ERC20 standard approve. To prevent attack vectors like the one described here and discussed here, clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. We consider this as a very low severity issue.

Testing Process

The **BC1 - Blockchain Pioneers** team reorganized the source code provided and added a standard environment to compile, tests and analyze contracts.

Lint

Lint failing

Lint failed, but they are all warnings on time-based decisions and using low level calls and assembly. These methods are required to make signatures and delegate working so they are not critical.

```
contracts/origin/contract/DeCashToken.sol
  767:17  warning  Avoid to make time-based decisions in your business logic  not-rely-on-time

contracts/origin/contract/DeCashUpgrade.sol
   94:13  warning  Avoid to make time-based decisions in your business logic  not-rely-on-time
  169:56  warning  Avoid to make time-based decisions in your business logic  not-rely-on-time
  195:62  warning  Avoid to make time-based decisions in your business logic  not-rely-on-time
  233:33  warning  Avoid to make time-based decisions in your business logic  not-rely-on-time

contracts/origin/lib/Address.sol
  153:13  warning  Avoid to use low level calls  avoid-low-level-calls

contracts/origin/lib/DeCashSignature.sol
  134:9  warning  Avoid to use inline assembly. It is acceptable only in rare cases  no-inline-assembly

contracts/origin/Migrations.sol
  6:20  warning  Variable name must be in mixedCase  var-name-mixedcase

✖ 8 problems (0 errors, 8 warnings)
```

Test

Test passing

Tests have not been provided by the **DeCash** team, but they have been written from the **BC1 - Blockchain Pioneers** team to try forcing a possible criticity.
Tests are in test folder.

```
  Contract: BC1 Tests
    testing network behaviours
      before token initialization
        check storage
          ✓ should have 0 as current version (71ms)
      after token initialization
        check storage
          ✓ should have 1 as current version (60ms)
        check delegatecall view
          ✓ has a name
```

```
          ✓ has a symbol (51ms)
          ✓ has an amount of decimals (48ms)
        check multisignature
          with 1 required signature
            owner can mint
              ✓ increments totalSupply (75ms)
              ✓ increments thirdParty balance (110ms)
              ✓ emits Transfer event
          with 2 required signature
            when the first user calls
              ✓ emit OperationUpvoted event but not OperationPerformed
              ✓ doesn't increment totalSupply (117ms)
              when the second user calls
                ✓ emit OperationUpvoted event and OperationPerformed event
                ✓ increments totalSupply (73ms)
        check signed token transfer
          ✓ success using personal sign standard (911ms)
          ✓ success using personal sign standard with zero fee (290ms)
          ✓ revert with signature with expired deadline (551ms)
          ✓ revert if re-use signature (396ms)
        estimate gas for transferMany
          with less than 100 receivers
            ✓ gas should remain under 8M (2803ms)
      testing proxy upgrade
        try to reinitialize the proxy
          ✓ fail (300ms)
        trying to upgrade proxy from owner
          ✓ revert (270ms)
      testing storage upgrade
        upgrade contract (testing DeCashRole)
          ✓ should success (827ms)
        Upgrade contract (testing DeCashToken)
          ✓ should success (1214ms)


  21 passing (2m)
```

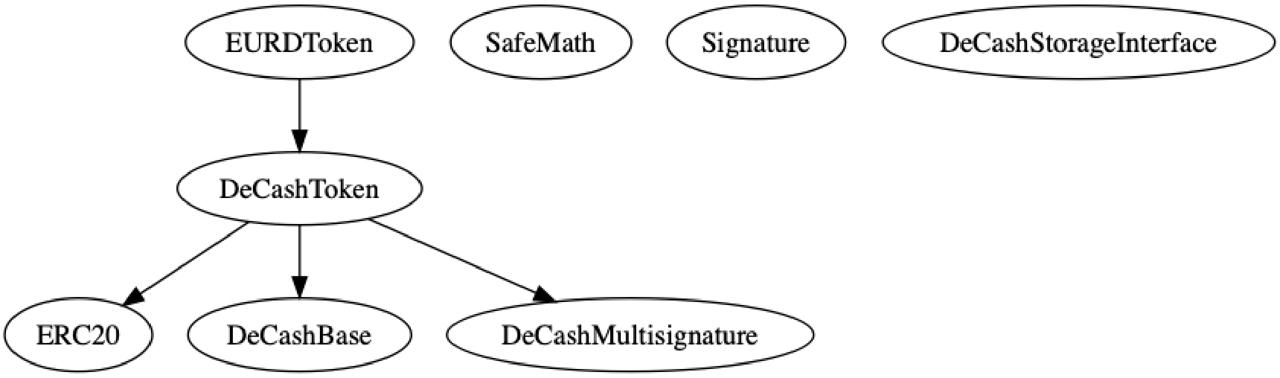
Structure Analysis

Inheritance Tree Graph

The **BC1 - Blockchain Pioneers** team built an inheritance tree graph for the smart contracts provided.

Graphs can be found at https://github.com/ConKreter/Decash-SmartContract/tree/audit/BC1/analysis/inheritance-tree.

Example: EURDToken

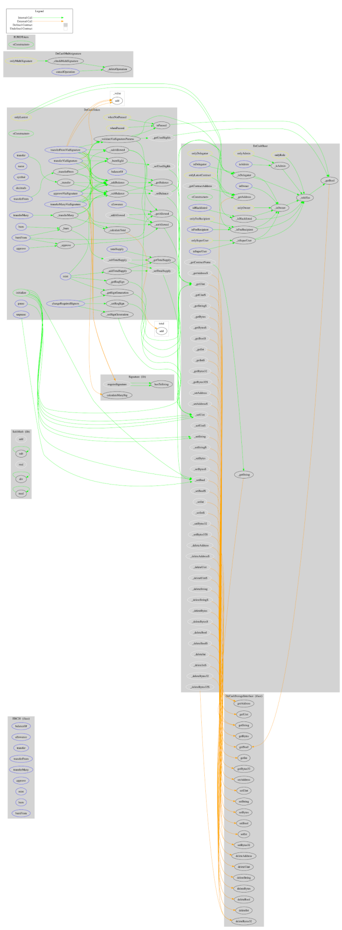


Control Flow Graph

The **BC1 - Blockchain Pioneers** team built a control flow graph for the smart contracts provided.

Graphs can be found at
https://github.com/ConKreter/Decash-SmartContract/tree/audit/BC1/analysis/control-flow.

Example: EURDToken



File Description Table

The **BC1 - Blockchain Pioneer**s team built a description table for the smart contracts provided.

Tables can be found at https://github.com/ConKreter/Decash-SmartContract/tree/audit/BC1/analysis/description-table.

UML

The **BC1 - Blockchain Pioneer**s team built a UML for the smart contracts provided.

UMLs can be found at https://github.com/ConKreter/Decash-SmartContract/tree/audit/BC1/analysis/uml.

Example: EURDToken

Code Analysis

DeCashStorage

Source code: contracts/contract/DeCashStorage.sol

This contract is credited by Rocket Pool RocketStorage.sol.
It represents persistent storage and will be used in all the below contracts.

The DeCashStorage contract is used as a key/value data set where the key is the **kekkak256** of the **abi.encodePacked** composited string and the value is a bool/string/address/uint (etc.).

The getter/setter/deleter methods are described below.

```
contract DeCashStorage is DeCashStorageInterface {
```

Constructor

```
constructor() {
   _boolStorage[
       keccak256(abi.encodePacked("access.role", "owner", msg.sender))
   ] = true;
}
```

The contract constructor stores a boolean as **"access.role" "owner"** for message sender.

Properties

```
mapping(bytes32 => uint256) private _uIntStorage;
mapping(bytes32 => string) private _stringStorage;
mapping(bytes32 => address) private _addressStorage;
mapping(bytes32 => bytes) private _bytesStorage;
mapping(bytes32 => bool) private _boolStorage;
mapping(bytes32 => int256) private _intStorage;
mapping(bytes32 => bytes32) private _bytes32Storage;
```

These are all the storage types.

Modifiers

```
modifier onlyLatestDeCashNetworkContract() {
   if (
       _boolStorage[
           keccak256(abi.encodePacked("contract.storage.initialised"))
       ] == true
   ) {
       require(
```

```
            _boolStorage[
                keccak256(abi.encodePacked("contract.exists", msg.sender))
            ],
            "Invalid or outdated network contract"
        );
    }
    _;
}
```

onlyLatestDeCashNetworkContract: Only allow access from the latest version of a contract after deployment. The owner and other contracts are only allowed to set the storage upon deployment to register the initial contracts/settings, afterwards their direct access is disabled. It checks that message sender is a valid contract that exists in storage.

Methods

```
function getAddress(bytes32 _key) external view override returns (address) {
   return _addressStorage[_key];
}

function getUint(bytes32 _key) external view override returns (uint256) {
   return _uIntStorage[_key];
}

function getString(bytes32 _key)
   external
   view
   override
   returns (string memory)
{
   return _stringStorage[_key];
}

function getBytes(bytes32 _key)
   external
   view
   override
   returns (bytes memory)
{
   return _bytesStorage[_key];
}

function getBool(bytes32 _key) external view override returns (bool) {
   return _boolStorage[_key];
}

function getInt(bytes32 _key) external view override returns (int256) {
   return _intStorage[_key];
}

function getBytes32(bytes32 _key) external view override returns (bytes32) {
   return _bytes32Storage[_key];
}

function setAddress(bytes32 _key, address _value)
   external
   override
   onlyLatestDeCashNetworkContract
{
   _addressStorage[_key] = _value;
}

function setUint(bytes32 _key, uint256 _value)
   external
   override
   onlyLatestDeCashNetworkContract
{
   _uIntStorage[_key] = _value;
}
```

```
function setString(bytes32 _key, string calldata _value)
   external
   override
   onlyLatestDeCashNetworkContract
{
   _stringStorage[_key] = _value;
}

function setBytes(bytes32 _key, bytes calldata _value)
   external
   override
   onlyLatestDeCashNetworkContract
{
   _bytesStorage[_key] = _value;
}

function setBool(bytes32 _key, bool _value)
   external
   override
   onlyLatestDeCashNetworkContract
{
   _boolStorage[_key] = _value;
}

function setInt(bytes32 _key, int256 _value)
   external
   override
   onlyLatestDeCashNetworkContract
{
   _intStorage[_key] = _value;
}

function setBytes32(bytes32 _key, bytes32 _value)
   external
   override
   onlyLatestDeCashNetworkContract
{
   _bytes32Storage[_key] = _value;
}

function deleteAddress(bytes32 _key)
   external
   override
   onlyLatestDeCashNetworkContract
{
   delete _addressStorage[_key];
}

function deleteUint(bytes32 _key)
   external
   override
   onlyLatestDeCashNetworkContract
{
   delete _uIntStorage[_key];
}

function deleteString(bytes32 _key)
   external
   override
   onlyLatestDeCashNetworkContract
{
   delete _stringStorage[_key];
}

function deleteBytes(bytes32 _key)
   external
   override
   onlyLatestDeCashNetworkContract
{
   delete _bytesStorage[_key];
}

function deleteBool(bytes32 _key)
   external
   override
   onlyLatestDeCashNetworkContract
{
   delete _boolStorage[_key];
}

function deleteInt(bytes32 _key)
```

```
    external
    override
    onlyLatestDeCashNetworkContract
{
    delete _intStorage[_key];
}

function deleteBytes32(bytes32 _key)
    external
    override
    onlyLatestDeCashNetworkContract
{
    delete _bytes32Storage[_key];
}
```

These are all the storage getters/setters/deleters.

DeCashBase

Source code: [contracts/contract/DeCashBase.sol]

This contract is credited by Rocket Pool [RocketBase.sol].
It represents base settings/modifiers for all the below contracts.

```
abstract contract DeCashBase {
```

Constructor

```
constructor(address _decashStorageAddress) {
    _decashStorage = DeCashStorageInterface(_decashStorageAddress);
}
```

The contract constructor sets the main DeCashStorage address.

Properties

```
uint8 public version;

DeCashStorageInterface internal _decashStorage = DeCashStorageInterface(0);
```

These define a uint version for the contract and the main storage contract where primary persistent storage is maintained.

Modifiers

```
modifier onlyLatestContract(
    string memory _contractName,
    address _contractAddress
) {
    require(
        _contractAddress ==
```

```
            _getAddress(
                keccak256(
                    abi.encodePacked("contract.address", _contractName)
                )
            ),
        "Invalid or outdated contract"
    );
    _;
}
```

onlyLatestContract: Throws if called by any sender that doesn't match one of the supplied contracts or is the latest version of that contract.

```
modifier onlyOwner() {
   require(_isOwner(msg.sender), "Account is not the owner");
   _;
}
modifier onlyAdmin() {
   require(_isAdmin(msg.sender), "Account is not an admin");
   _;
}
modifier onlySuperUser() {
   require(_isSuperUser(msg.sender), "Account is not a super user");
   _;
}
modifier onlyDelegator(address _address) {
   require(_isDelegator(_address), "Account is not a delegator");
   _;
}
modifier onlyFeeRecipient(address _address) {
   require(_isFeeRecipient(_address), "Account is not a fee recipient");
   _;
}
modifier onlyRole(string memory _role) {
   require(_roleHas(_role, msg.sender), "Account does not match the role");
   _;
}
```

onlyOwner, **onlyAdmin**, **onlySuperUser**, **onlyDelegator**, **onlyFeeRecipient**, **onlyRole**: Throws if called by any sender that doesn't have the specified role.

Methods

```
function _getContractAddress(string memory _contractName)
   internal
   view
   returns (address)
{
   address contractAddress =
       _getAddress(
           keccak256(abi.encodePacked("contract.address", _contractName))
       );

   require(contractAddress != address(0x0), "Contract not found");
```

```
    return contractAddress;
}
```

Get the address of a network contract by name.

Suggestion:

- ⚠️ This is an internal function and seems to be unused. It is here for future purpose only as explained by the DeCash team.

```
function _getContractName(address _contractAddress)
    internal
    view
    returns (string memory)
{
    string memory contractName =
        _getString(
            keccak256(abi.encodePacked("contract.name", _contractAddress))
        );

    require(
        keccak256(abi.encodePacked(contractName)) !=
            keccak256(abi.encodePacked("")),
        "Contract not found"
    );

    return contractName;
}
```

Get the name of a network contract by address.

Suggestion:

- ⚠️ This is an internal function and seems to be unused. It is here for future purpose only as explained by the DeCash team.

```
function _roleHas(string memory _role, address _address)
    internal
    view
    returns (bool)
{
    return
        _getBool(
            keccak256(abi.encodePacked("access.role", _role, _address))
        );
}

function _isOwner(address _address) internal view returns (bool) {
    return _roleHas("owner", _address);
}

function _isAdmin(address _address) internal view returns (bool) {
    return _roleHas("admin", _address);
}

function _isSuperUser(address _address) internal view returns (bool) {
    return _roleHas("admin", _address) || _isOwner(_address);
```

```
}

function _isDelegator(address _address) internal view returns (bool) {
    return _roleHas("delegator", _address) || _isOwner(_address);
}

function _isFeeRecipient(address _address) internal view returns (bool) {
    return _roleHas("fee", _address) || _isOwner(_address);
}

function _isBlacklisted(address _address) internal view returns (bool) {
    return _roleHas("blacklisted", _address) && !_isOwner(_address);
}
```

_roleHas: Allows to check if an address has a role or not.

_isOwner, **_isAdmin**, **_isSuperUser**, **_isDelegator**, **_isFeeRecipient**, **_isBlacklisted**: These are shortcuts to the _roleHas method with additional checks.

Suggestion:

- ⚠️ Some of these methods check multiple conditions and may be confusing for users to understand. For instance the "_isFeeRecipient" method returns true also if the address has not the "fee" role but the "owner" role. It is not explicit in method name. It could be simple to read if these methods had been used with logical operators like:

  ```
  _isFeeRecipient(_address) || _isOwner(_address)
  ```

  This is not an issue on the contract behaviour.

```
function isOwner(address _address) external view returns (bool) {
    return _isOwner(_address);
}

function isAdmin(address _address) external view returns (bool) {
    return _isAdmin(_address);
}

function isSuperUser(address _address) external view returns (bool) {
    return _isSuperUser(_address);
}

function isDelegator(address _address) external view returns (bool) {
    return _isDelegator(_address);
}

function isFeeRecipient(address _address) external view returns (bool) {
    return _isFeeRecipient(_address);
}

function isBlacklisted(address _address) external view returns (bool) {
    return _isBlacklisted(_address);
}
```

isOwner, **isAdmin**, **isSuperUser**, **isDelegator**, **isFeeRecipient**, **isBlacklisted**: These are external methods to the shortcuts above.

Suggestion:

- ⚠️ They internally use the above methods who check multiple conditions and may be confusing for users to understand. For instance the “isFeeRecipient” method returns true also if the address has not the “fee” role but the “owner” role. It is not explicit in method name. It could be simple to read if these methods had been used with logical operators like:

  ```
  isFeeRecipient(_address) || isOwner(_address)
  ```

  This is not an issue on the contract behaviour.

```
function _getAddress(bytes32 _key) internal view returns (address) {
   return _decashStorage.getAddress(_key);
}

function _getUint(bytes32 _key) internal view returns (uint256) {
   return _decashStorage.getUint(_key);
}

function _getString(bytes32 _key) internal view returns (string memory) {
   return _decashStorage.getString(_key);
}

function _getBytes(bytes32 _key) internal view returns (bytes memory) {
   return _decashStorage.getBytes(_key);
}

function _getBool(bytes32 _key) internal view returns (bool) {
   return _decashStorage.getBool(_key);
}

function _getInt(bytes32 _key) internal view returns (int256) {
   return _decashStorage.getInt(_key);
}

function _getBytes32(bytes32 _key) internal view returns (bytes32) {
   return _decashStorage.getBytes32(_key);
}

function _getAddressS(string memory _key) internal view returns (address) {
   return _decashStorage.getAddress(keccak256(abi.encodePacked(_key)));
}

function _getUintS(string memory _key) internal view returns (uint256) {
   return _decashStorage.getUint(keccak256(abi.encodePacked(_key)));
}

function _getStringS(string memory _key)
   internal
   view
   returns (string memory)
{
   return _decashStorage.getString(keccak256(abi.encodePacked(_key)));
}

function _getBytesS(string memory _key)
   internal
   view
   returns (bytes memory)
{
   return _decashStorage.getBytes(keccak256(abi.encodePacked(_key)));
}

function _getBoolS(string memory _key) internal view returns (bool) {
   return _decashStorage.getBool(keccak256(abi.encodePacked(_key)));
}

function _getIntS(string memory _key) internal view returns (int256) {
   return _decashStorage.getInt(keccak256(abi.encodePacked(_key)));
}

function _getBytes32S(string memory _key) internal view returns (bytes32) {
   return _decashStorage.getBytes32(keccak256(abi.encodePacked(_key)));
```

```
}

/// @dev Storage set methods
function _setAddress(bytes32 _key, address _value) internal {
   _decashStorage.setAddress(_key, _value);
}

function _setUint(bytes32 _key, uint256 _value) internal {
   _decashStorage.setUint(_key, _value);
}

function _setString(bytes32 _key, string memory _value) internal {
   _decashStorage.setString(_key, _value);
}

function _setBytes(bytes32 _key, bytes memory _value) internal {
   _decashStorage.setBytes(_key, _value);
}

function _setBool(bytes32 _key, bool _value) internal {
   _decashStorage.setBool(_key, _value);
}

function _setInt(bytes32 _key, int256 _value) internal {
   _decashStorage.setInt(_key, _value);
}

function _setBytes32(bytes32 _key, bytes32 _value) internal {
   _decashStorage.setBytes32(_key, _value);
}

function _setAddressS(string memory _key, address _value) internal {
   _decashStorage.setAddress(keccak256(abi.encodePacked(_key)), _value);
}

function _setUintS(string memory _key, uint256 _value) internal {
   _decashStorage.setUint(keccak256(abi.encodePacked(_key)), _value);
}

function _setStringS(string memory _key, string memory _value) internal {
   _decashStorage.setString(keccak256(abi.encodePacked(_key)), _value);
}

function _setBytesS(string memory _key, bytes memory _value) internal {
   _decashStorage.setBytes(keccak256(abi.encodePacked(_key)), _value);
}

function _setBoolS(string memory _key, bool _value) internal {
   _decashStorage.setBool(keccak256(abi.encodePacked(_key)), _value);
}

function _setIntS(string memory _key, int256 _value) internal {
   _decashStorage.setInt(keccak256(abi.encodePacked(_key)), _value);
}

function _setBytes32S(string memory _key, bytes32 _value) internal {
   _decashStorage.setBytes32(keccak256(abi.encodePacked(_key)), _value);
}

/// @dev Storage delete methods
function _deleteAddress(bytes32 _key) internal {
   _decashStorage.deleteAddress(_key);
}

function _deleteUint(bytes32 _key) internal {
   _decashStorage.deleteUint(_key);
}

function _deleteString(bytes32 _key) internal {
   _decashStorage.deleteString(_key);
}

function _deleteBytes(bytes32 _key) internal {
   _decashStorage.deleteBytes(_key);
}

function _deleteBool(bytes32 _key) internal {
   _decashStorage.deleteBool(_key);
}

function _deleteInt(bytes32 _key) internal {
```

```
    _decashStorage.deleteInt(_key);
}

function _deleteBytes32(bytes32 _key) internal {
    _decashStorage.deleteBytes32(_key);
}

function _deleteAddressS(string memory _key) internal {
    _decashStorage.deleteAddress(keccak256(abi.encodePacked(_key)));
}

function _deleteUintS(string memory _key) internal {
    _decashStorage.deleteUint(keccak256(abi.encodePacked(_key)));
}

function _deleteStringS(string memory _key) internal {
    _decashStorage.deleteString(keccak256(abi.encodePacked(_key)));
}

function _deleteBytesS(string memory _key) internal {
    _decashStorage.deleteBytes(keccak256(abi.encodePacked(_key)));
}

function _deleteBoolS(string memory _key) internal {
    _decashStorage.deleteBool(keccak256(abi.encodePacked(_key)));
}

function _deleteIntS(string memory _key) internal {
    _decashStorage.deleteInt(keccak256(abi.encodePacked(_key)));
}

function _deleteBytes32S(string memory _key) internal {
    _decashStorage.deleteBytes32(keccak256(abi.encodePacked(_key)));
}
```

These are all the storage getters/setters/deleters to handle the DeCashStorage persistent data.

Suggestion:

- ⚠️ All the methods ending with a capped “S” (like **getAddressS**, **setAddressS**, **deleteAddressS**…) are useful methods to access the storage with the raw _key instead of using the **web3.utils.soliditySha3** method to retrieve the hashed one. But they seem to be unused in derived contracts. They are here for future purpose only as explained by the DeCash team.

DeCashRole

Source code: contracts/contract/DeCashRole.sol

This contract is credited by Rocket Pool RocketRole.sol.
It represents a Role Based Access Control for the DeCash contracts.

```
contract DeCashRole is DeCashBase, DeCashRoleInterface {
```

Constructor

```
constructor(address _decashStorageAddress)
    DeCashBase(_decashStorageAddress)
{
    version = 1;
}
```

The contract constructor sets the main DeCashStorage address and the contract version.

Events

```
event RoleAdded(bytes32 indexed role, address indexed to);
event RoleRemoved(bytes32 indexed role, address indexed to);
event OwnershipTransferred(address indexed from, address indexed to);
```

These events will be emitted after role changes or ownership transfers.

Methods

```
function transferOwnership(address _newOwner)
    external
    override
    onlyLatestContract("role", address(this))
    onlyOwner
{
    require(_newOwner != address(0x0), "The new owner address is invalid");
    require(
        _newOwner != msg.sender,
        "The new owner address must not be the existing owner address"
    );

    _deleteBool(
        keccak256(abi.encodePacked("access.role", "owner", msg.sender))
    );

    _setBool(
        keccak256(abi.encodePacked("access.role", "owner", _newOwner)),
        true
    );

    emit OwnershipTransferred(msg.sender, _newOwner);
}
```

The **transferOwnership** method allows to transfer contract ownership to a new owner.

```
function addRole(string memory _role, address _address)
    external
    override
    onlyLatestContract("role", address(this))
    onlySuperUser
{
    require(
        keccak256(abi.encodePacked(_role)) !=
            keccak256(abi.encodePacked("owner")),
        "The owner role cannot be added to an address"
```

```
    );

    require(_address != address(0x0), "The address is invalid");
    require(
        !_getBool(
            keccak256(abi.encodePacked("access.role", _role, _address))
        ),
        "The address already has access to this role"
    );

    _setBool(
        keccak256(abi.encodePacked("access.role", _role, _address)),
        true
    );

    emit RoleAdded(keccak256(abi.encodePacked(_role)), _address);
}
```

The **addRole** method allows to add roles to an address.

```
function removeRole(string memory _role, address _address)
    external
    override
    onlyLatestContract("role", address(this))
    onlySuperUser
{
    require(
        !_roleHas("owner", _address),
        "Roles cannot be removed from the owner address"
    );

    require(
        _getBool(
            keccak256(abi.encodePacked("access.role", _role, _address))
        ),
        "The address does not have access to this role"
    );

    _deleteBool(
        keccak256(abi.encodePacked("access.role", _role, _address))
    );

    emit RoleRemoved(keccak256(abi.encodePacked(_role)), _address);
}
```

The **removeRole** method allows to remove roles from an address (other than the owner).

Suggestion:

- ⚠️ Use “_isOwner” instead of “_roleHas(‘owner’)” in require.
- ⚠️ Allow to remove roles also from the owner. By the code, it is impossible to remove any role from the owner but if for some reason it will be required, it won’t be possible.

DeCashProxy

Source code: [contracts/contract/DeCashProxy.sol]

This contract is inherited by OpenZeppelin Proxy.sol and credited by OpenZeppelin UpgradeableProxy.sol. It implements the EIP-1967: Standard Proxy Storage Slots.

This contract implements an upgradeable proxy. It is upgradeable because calls are delegated to an implementation address that can be changed. This address is stored in storage in the location specified by EIP1967, so that it doesn't conflict with the storage layout of the implementation behind the proxy.

```
contract DeCashProxy is DeCashBase, Proxy {
```

Constructor

```
constructor(address _decashStorageAddress)
   DeCashBase(_decashStorageAddress)
{
   assert(
       _IMPLEMENTATION_SLOT ==
           bytes32(uint256(keccak256("eip1967.proxy.implementation")) - 1)
   );
   version = 1;
}
```

The contract constructor sets the main DeCashStorage address and contract version. It also checks invariants.

Events

```
event ProxyInitiated(address indexed implementation);
event ProxyUpgraded(address indexed implementation);
```

These events will be emitted after proxy initiated or upgraded.

Properties

```
bytes32 private constant _IMPLEMENTATION_SLOT =
   0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;
```

Storage slot with the address of the current implementation. This is the keccak-256 hash of "eip1967.proxy.implementation" subtracted by 1, and is validated in the constructor.

Methods

```
function _implementation() internal view override returns (address impl) {
    bytes32 slot = _IMPLEMENTATION_SLOT;
    assembly {
        impl := sload(slot)
    }
}
```

The **_implementation** method returns the current implementation address. This overrides the Proxy method and will be used to delegate calls to the proxied contract.

```
function upgrade(address _address)
    public
    onlyLatestContract("upgrade", msg.sender)
{
    _setImplementation(_address);

    emit ProxyUpgraded(_address);
}

function initialize(address _address) external onlyOwner {
    require(
        !_getBool(keccak256(abi.encodePacked("proxy.init", address(this)))),
        "Proxy already initialized"
    );

    _setImplementation(_address);
    _setBool(keccak256(abi.encodePacked("proxy.init", address(this)), true);

    emit ProxyInitiated(_address);
}
```

The **initialize** and **upgrade** methods, respectively, set the initial contract implementation and upgrade it using the below _setImplementation method. They also emit the above events.

Suggestion:

- ⚠️ Use the “external” visibility modifier for functions never called from the contract via internal call.

```
function _setImplementation(address _address) private {
    require(Address.isContract(_address), "address is not a contract");

    bytes32 slot = _IMPLEMENTATION_SLOT;

    assembly {
        sstore(slot, _address)
```

```
    }
}
```

The **_setImplementation** method stores a new address in the EIP1967 implementation slot.

DeCashUpgrade

Source code: contracts/contract/DeCashUpgrade.sol

This contract is credited by Rocket Pool RocketUpgrade.sol.

This contract handles network contracts upgrades.

```
contract DeCashUpgrade is DeCashBase, DeCashUpgradeInterface {
```

Constructor

```
constructor(address _decashStorageAddress)
    DeCashBase(_decashStorageAddress)
{
    version = 1;
}
```

The contract constructor sets the main DeCashStorage address and contract version.

Events

```
event ContractUpgraded(
    bytes32 indexed name,
    address indexed oldAddress,
    address indexed newAddress,
    uint256 time
);
event ContractAdded(
    bytes32 indexed name,
    address indexed newAddress,
    uint256 time
);
event ABIUpgraded(bytes32 indexed name, uint256 time);
event ABIAdded(bytes32 indexed name, uint256 time);
```

These events will be emitted after contract or ABI added or upgraded.

Methods

```
function upgradeContract(
    string memory _name,
    address _contractAddress,
    string memory _contractAbi
)
    external
    override
    onlyLatestContract("upgrade", address(this))
    onlySuperUser
{
    bytes32 nameHash = keccak256(abi.encodePacked(_name));
    require(nameHash != keccak256(abi.encodePacked("proxy")), "Cannot upgrade
proxy contracts");

    address oldContractAddress =
        _getAddress(keccak256(abi.encodePacked("contract.address", _name)));
    require(oldContractAddress != address(0x0), "Contract does not exist");

    require(_contractAddress != address(0x0), "Invalid contract address");
    require(
        _contractAddress != oldContractAddress,
        "The contract address cannot be set to its current address"
    );

    _setBool(
        keccak256(abi.encodePacked("contract.exists", _contractAddress)),
        true
    );
    _setString(
        keccak256(abi.encodePacked("contract.name", _contractAddress)),
        _name
    );
    _setAddress(
        keccak256(abi.encodePacked("contract.address", _name)),
        _contractAddress
    );
    _setString(
        keccak256(abi.encodePacked("contract.abi", _name)),
        _contractAbi
    );

    _deleteString(
        keccak256(abi.encodePacked("contract.name", oldContractAddress))
    );
    _deleteBool(
        keccak256(abi.encodePacked("contract.exists", oldContractAddress))
    );
```

```
    emit ContractUpgraded(
        nameHash,
        oldContractAddress,
        _contractAddress,
        block.timestamp
    );

    if (nameHash == keccak256(abi.encodePacked("token"))) {
        DeCashProxyInterface proxy =
            DeCashProxyInterface(
                _getAddress(
                    keccak256(abi.encodePacked("contract.address", "proxy"))
                )
            );
        proxy.upgrade(_contractAddress);
    }
}
```

The **upgradeContract** method allows to update the storage with an upgraded version of the contracts. If the token contract is being updated, this method also upgrades the proxy implementation slot. Proxy contract can't be updated once deployed.

```
function addContract(
    string memory _name,
    address _contractAddress,
    string memory _contractAbi
)
    external
    override
    onlyLatestContract("upgrade", address(this))
    onlySuperUser
{
    bytes32 nameHash = keccak256(abi.encodePacked(_name));
    require(
        nameHash != keccak256(abi.encodePacked("")),
        "Invalid contract name"
    );
    require(
        _getAddress(
            keccak256(abi.encodePacked("contract.address", _name))
        ) == address(0x0),
        "Contract name is already in use"
    );

    string memory existingAbi =
        _getString(keccak256(abi.encodePacked("contract.abi", _name)));
    require(
        keccak256(abi.encodePacked(existingAbi)) ==
            keccak256(abi.encodePacked("")),
        "Contract name is already in use"
    );
```

```
    require(_contractAddress != address(0x0), "Invalid contract address");
    require(
        !_getBool(
            keccak256(abi.encodePacked("contract.exists", _contractAddress))
        ),
        "Contract address is already in use"
    );

    _setBool(
        keccak256(abi.encodePacked("contract.exists", _contractAddress)),
        true
    );
    _setString(
        keccak256(abi.encodePacked("contract.name", _contractAddress)),
        _name
    );
    _setAddress(
        keccak256(abi.encodePacked("contract.address", _name)),
        _contractAddress
    );
    _setString(
        keccak256(abi.encodePacked("contract.abi", _name)),
        _contractAbi
    );

    emit ContractAdded(nameHash, _contractAddress, block.timestamp);
}
```

The **addContract** method allows adding a new contract to the network.

```
function upgradeABI(string memory _name, string memory _contractAbi)
    external
    override
    onlyLatestContract("upgrade", address(this))
    onlySuperUser
{
    string memory existingAbi =
        _getString(keccak256(abi.encodePacked("contract.abi", _name)));
    require(
        keccak256(abi.encodePacked(existingAbi)) !=
            keccak256(abi.encodePacked("")),
        "ABI does not exist"
    );

    _setString(
        keccak256(abi.encodePacked("contract.abi", _name)),
        _contractAbi
    );

    emit ABIUpgraded(keccak256(abi.encodePacked(_name)), block.timestamp);
}

function addABI(string memory _name, string memory _contractAbi)
    external
    override
    onlyLatestContract("upgrade", address(this))
    onlySuperUser
```

```
{
    bytes32 nameHash = keccak256(abi.encodePacked(_name));
    require(
        nameHash != keccak256(abi.encodePacked("")),
        "Invalid ABI name"
    );
    require(
        _getAddress(
            keccak256(abi.encodePacked("contract.address", _name))
        ) == address(0x0),
        "ABI name is already in use"
    );

    string memory existingAbi =
        _getString(keccak256(abi.encodePacked("contract.abi", _name)));
    require(
        keccak256(abi.encodePacked(existingAbi)) ==
            keccak256(abi.encodePacked("")),
        "ABI name is already in use"
    );

    _setString(
        keccak256(abi.encodePacked("contract.abi", _name)),
        _contractAbi
    );

    emit ABIAdded(nameHash, block.timestamp);
}
```

The **upgradeABI** method upgrades a network contract ABI. The **addABI** method adds a new network contract ABI.

DeCashSignature

Source code: [contracts/lib/DeCashSignature.sol]()

This library is inspired by DreamTeam Group Limited [DreamTeamToken.sol]().

This is a utility library to handle signatures.

```
library Signature {
```

Enum

```
enum Std {typed, personal, stringHex}

enum Dest {transfer, transferFrom, transferMany, approve, approveAndCall}
```

These enums are used to choose a standard for signature validation and to explicit the method to call.

Constants

```
bytes public constant ETH_SIGNED_MESSAGE_PREFIX =
   "\x19Ethereum Signed Message:\n";

// `transferViaSignature`: keccak256(abi.encodePacked(address(this), from, to,
value, fee, deadline, sigId))
bytes32 public constant DEST_TRANSFER =
   keccak256(
       abi.encodePacked(
           "address Contract",
           "address Sender",
           "address Recipient",
           "uint256 Amount (last 2 digits are decimals)",
           "uint256 Fee Amount (last 2 digits are decimals)",
           "address Fee Address",
           "uint256 Expiration",
           "uint256 Signature ID"
       )
   );

// `transferManyViaSignature`: keccak256(abi.encodePacked(address(this), from,
to/value array, deadline, sigId))
bytes32 public constant DEST_TRANSFER_MANY =
   keccak256(
       abi.encodePacked(
           "address Contract",
           "address Sender",
           "bytes32 Recipient/Amount Array hash",
           "uint256 Fee Amount (last 2 digits are decimals)",
           "address Fee Address",
           "uint256 Expiration",
           "uint256 Signature ID"
       )
   );

// `transferFromViaSignature`: keccak256(abi.encodePacked(address(this), signer,
from, to, value, fee, deadline, sigId))
bytes32 public constant DEST_TRANSFER_FROM =
   keccak256(
       abi.encodePacked(
           "address Contract",
           "address Approved",
           "address From",
           "address Recipient",
           "uint256 Amount (last 2 digits are decimals)",
           "uint256 Fee Amount (last 2 digits are decimals)",
           "address Fee Address",
           "uint256 Expiration",
```

```
            "uint256 Signature ID"
        )
    );

// `approveViaSignature`: keccak256(abi.encodePacked(address(this), from,
spender, value, fee, deadline, sigId))
bytes32 public constant DEST_APPROVE =
    keccak256(
        abi.encodePacked(
            "address Contract",
            "address Approval",
            "address Recipient",
            "uint256 Amount (last 2 digits are decimals)",
            "uint256 Fee Amount (last 2 digits are decimals)",
            "address Fee Address",
            "uint256 Expiration",
            "uint256 Signature ID"
        )
    );

// `approveAndCallViaSignature`: keccak256(abi.encodePacked(address(this), from,
spender, value, extraData, fee, deadline, sigId))
bytes32 public constant DEST_APPROVE_AND_CALL =
    keccak256(
        abi.encodePacked(
            "address Contract",
            "address Approval",
            "address Recipient",
            "uint256 Amount (last 2 digits are decimals)",
            "bytes Data to Transfer",
            "uint256 Fee Amount (last 2 digits are decimals)",
            "address Fee Address",
            "uint256 Expiration",
            "uint256 Signature ID"
        )
    );
```

These constants are the keccak256 hash of the required method.

Methods

```
function hexToString(bytes32 sig) internal pure returns (bytes memory) {
    bytes memory str = new bytes(64);

    for (uint8 i = 0; i < 32; ++i) {
        str[2 * i] = bytes1(
            (uint8(sig[i]) / 16 < 10 ? 48 : 87) + uint8(sig[i]) / 16
        );
        str[2 * i + 1] = bytes1(
            (uint8(sig[i]) % 16 < 10 ? 48 : 87) + (uint8(sig[i]) % 16)
```

```
        );
    }

    return str;
}
```

The **hexToString** method is an utility costly function to encode bytes HEX representation as string.

```
function requireSignature(
    bytes32 _data,
    address _signer,
    bytes memory _sig,
    Std _sigStd,
    Dest _sigDest
) internal {
    bytes32 r;
    bytes32 s;
    uint8 v;

    assembly {
        // solium-disable-line security/no-inline-assembly
        r := mload(add(_sig, 32))
        s := mload(add(_sig, 64))
        v := byte(0, mload(add(_sig, 96)))
    }

    if (v < 27) v += 27;

    if (_sigStd == Std.typed) {
        bytes32 dest;

        if (_sigDest == Dest.transfer) {
            dest = DEST_TRANSFER;
        } else if (_sigDest == Dest.transferMany) {
            dest = DEST_TRANSFER_MANY;
        } else if (_sigDest == Dest.transferFrom) {
            dest = DEST_TRANSFER_FROM;
        } else if (_sigDest == Dest.approve) {
            dest = DEST_APPROVE;
        } else if (_sigDest == Dest.approveAndCall) {
            dest = DEST_APPROVE_AND_CALL;
        }

        // Typed signature. This is the most likely scenario to be used and accepted
        require(
            _signer ==
                ecrecover(
                    keccak256(abi.encodePacked(dest, _data)),
                    v,
                    r,
                    s
                ),
            "Invalid typed signature"
        );
    } else if (_sigStd == Std.personal) {
        // Ethereum signed message signature (Geth and Trezor)
        require(
```

```
            _signer ==
                ecrecover(
                    keccak256(
                        abi.encodePacked(
                            ETH_SIGNED_MESSAGE_PREFIX,
                            "32",
                            _data
                        )
                    ),
                    v,
                    r,
                    s
                ) || // Geth-adopted
                _signer ==
                ecrecover(
                    keccak256(
                        abi.encodePacked(
                            ETH_SIGNED_MESSAGE_PREFIX,
                            "\x20",
                            _data
                        )
                    ),
                    v,
                    r,
                    s
                ), // Trezor-adopted
            "Invalid personal signature"
        );
    } else {
        // == 2; Signed string hash signature (the most expensive but universal)
        require(
            _signer ==
                ecrecover(
                    keccak256(
                        abi.encodePacked(
                            ETH_SIGNED_MESSAGE_PREFIX,
                            "64",
                            hexToString(_data)
                        )
                    ),
                    v,
                    r,
                    s
                ) || // Geth
                _signer ==
                ecrecover(
                    keccak256(
                        abi.encodePacked(
                            ETH_SIGNED_MESSAGE_PREFIX,
                            "\x40",
                            hexToString(_data)
                        )
                    ),
                    v,
                    r,
                    s
                ), // Trezor
            "Invalid stringHex signature"
        );
```

```
    }
}
```

The **requireSignature** is an internal method that makes sure that the given signature corresponds to a given data and is made by `signer`. Throws otherwise. It utilizes different standards of message signing in Ethereum, as at the moment there is no single signing standard defined. For example, Metamask and Geth both support personal_sign standard, SignTypedData is only supported by Matamask, Trezor does not support "widely adopted" Ethereum personal_sign but rather personal_sign with fixed prefix and so on.

Note that it is always possible to forge any of these signatures using the private key, the problem is that third-party wallets must adopt a single standard for signing messages.

Params:

- **_data**: original data which had to be signed by `signer`
- **_signer**: account which made a signature
- **_sig**: signature made by `from`, which is the proof of `from`'s agreement with the above parameters
- **_sigStd**: chosen standard for signature validation. The signer must explicitly tell which standard they use
- **_sigDest**: for which type of action this signature was made for

```
function calculateManySig(address[] memory _tos, uint256[] memory _values)
    internal
    pure
    returns (bytes32)
{
    bytes32 tv = keccak256(abi.encodePacked(_tos[0], _values[0]));

    uint256 ln = _tos.length;

    for (uint8 x = 1; x < ln; x++) {
        tv = keccak256(abi.encodePacked(tv, _tos[x], _values[x]));
    }

    return tv;
}
```

The **calculateManySig** method is an utility function to calculate the signature of the array of recipient/value pairs to use in transferMany.

DeCashMultisignature

Source code: [contracts/contract/DeCashMultisignature.sol]()

This contract seems inspired by BitClave [Multiownable.sol]().

This contract checks multiple confirmations before performing an operation.

```
abstract contract DeCashMultisignature {
```

Events

```
event RequiredSignerChanged(
    uint256 newRequiredSignature,
    uint256 generation
);
event OperationCreated(bytes32 operation, address proposer);
event OperationUpvoted(bytes32 operation, address voter);
event OperationPerformed(bytes32 operation, address performer);
event OperationCancelled(bytes32 operation, address performer);
```

These events will be emitted after the number of signatures changed or after actions on operations.

Properties

```
bytes32[] public allOperations;
mapping(bytes32 => uint256) public allOperationsIndicies;
mapping(bytes32 => uint256) public votesCountByOperation;
mapping(bytes32 => address) public firstByOperation;
mapping(bytes32 => mapping(address => uint8)) public votesOwnerByOperation;
mapping(bytes32 => address[]) public votesIndicesByOperation;

uint256 public signerGeneration;
address internal _insideCallSender;
uint256 internal _insideCallCount;
```

These properties allow to store information about operations and their statuses.
The last ones are utility variables to be used in the below functions and modifiers.

Modifiers

```
modifier onlyMultiSignature(uint256 _howMany, uint256 _generation) {
    if (_checkMultiSignature(_howMany, _generation)) {
        bool update = (_insideCallSender == address(0));
        if (update) {
            _insideCallSender = msg.sender;
            _insideCallCount = _howMany;
```

```
        }

        _;

        if (update) {
            _insideCallSender = address(0);
            _insideCallCount = 0;
        }
    }
}
```

onlyMultiSignature: Allows to perform a method only after many owners call it with the same arguments.

Methods

```
function _checkMultiSignature(uint256 _howMany, uint256 _generation)
    internal
    returns (bool)
{
    if (_howMany < 2) return true;

    if (_insideCallSender == msg.sender) {
        require(_howMany <= _insideCallCount, "howMany > _insideCallCount");
        return true;
    }

    bytes32 operation = keccak256(abi.encodePacked(msg.data, _generation));

    uint256 operationVotesCount = votesCountByOperation[operation] + 1;
    votesCountByOperation[operation] = operationVotesCount;

    if (firstByOperation[operation] == address(0)) {
        firstByOperation[operation] = msg.sender;

        allOperationsIndicies[operation] = allOperations.length;
        allOperations.push(operation);

        emit OperationCreated(operation, msg.sender);
    } else {
        require(
            votesOwnerByOperation[operation][msg.sender] == 0,
            "[operation][msg.sender] != 0"
        );
    }

    votesIndicesByOperation[operation].push(msg.sender);
    votesOwnerByOperation[operation][msg.sender] = 1;

    emit OperationUpvoted(operation, msg.sender);
```

```
    if (operationVotesCount < _howMany) return false;

    _deleteOperation(operation);

    emit OperationPerformed(operation, msg.sender);

    return true;
}
```

The **checkMultiSignature** is an internal method used in the above modifiers to create/upvote/perform the method where the modifier will be applied.

```
function _deleteOperation(bytes32 operation) internal {
    uint256 index = allOperationsIndicies[operation];
    if (index < allOperations.length - 1) {
        // Not last
        allOperations[index] = allOperations[allOperations.length - 1];
        allOperationsIndicies[allOperations[index]] = index;
    }

    delete allOperations[allOperations.length - 1];
    delete allOperationsIndicies[operation];
    delete votesCountByOperation[operation];
    delete firstByOperation[operation];

    uint8 x;
    uint256 ln = votesIndicesByOperation[operation].length;

    for (x = 0; x < ln; x++) {
        delete votesOwnerByOperation[operation][
            votesIndicesByOperation[operation][x]
        ];
    }

    for (x = 0; x < ln; x++) {
        votesIndicesByOperation[operation].pop();
    }
}
```

The **deleteOperation** is an internal method used to delete a previously inserted operation.

```
function cancelOperation(bytes32 operation) external {
    require(votesCountByOperation[operation] > 0, "Operation not
found");

    _deleteOperation(operation);

```

```
    emit OperationCancelled(operation, msg.sender);
}
```

The **cancelOperation** is a public method that allows owners to change their mind by cancelling operations.

DeCashToken

Source code: contracts/contract/DeCashToken.sol

It represents the Token contract from where any other currency will be inherited.

```
contract DeCashToken is DeCashBase, DeCashMultisignature, ERC20 {
```

Constructor

```
constructor(address _decashStorageAddress)
   DeCashBase(_decashStorageAddress)
{
   version = 1;
}
```

The contract constructor sets the main DeCashStorage address and the contract version.

Properties

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

These define the ERC20 standard details.

Events

```
event Paused(address indexed from);
event Unpaused(address indexed from);
event Transfer(address indexed from, address indexed to, uint256 value);
event Approval(
   address indexed owner,
   address indexed spender,
   uint256 value
);
```

The **Transfer** and **Approval** events are the standard ERC20 events.
The **Paused** and **Unpaused** events are emitted when the contracts are set to pause or unpause as described below.

Modifiers

```
modifier onlyLastest {
    require(
        address(this) ==
            _getAddress(
                keccak256(abi.encodePacked("contract.address", "token"))
            ) ||
            address(this) ==
            _getAddress(
                keccak256(abi.encodePacked("contract.address", "proxy"))
            ),
        "Invalid or outdated contract"
    );
    _;
}
```

onlyLatest: Checks if the contract calling is the latest version stored in the DeCashStorage.

```
modifier whenNotPaused {
   require(!isPaused(), "Contract is paused");
   _;
}
modifier whenPaused {
   require(isPaused(), "Contract is not paused");
   _;
}
```

whenNotPaused, **whenPaused**: Throws if contract is, respectively, not paused or paused.

Methods

```
function initialize(
   string memory _tokenName,
   string memory _tokenSymbol,
   uint8 _tokenDecimals
) public onlyOwner {
   uint256 currentVersion =
       _getUint(keccak256(abi.encodePacked("token.version", _tokenName)));

   if (currentVersion == 0) {
       _name = _tokenName;
       _symbol = _tokenSymbol;
       _decimals = _tokenDecimals;

       _setString(keccak256(abi.encodePacked("token.name", _name)), _name);
       _setString(
           keccak256(abi.encodePacked("token.symbol", _name)),
           _symbol
       );
       _setUint(
           keccak256(abi.encodePacked("token.decimals", _name)),
           _decimals
```

```
        );
        _setBool(
            keccak256(abi.encodePacked("contract.paused", _name)),
            false
        );
        _setUint(keccak256(abi.encodePacked("mint.reqSign", _name)), 1);
    }

    if (currentVersion != version) {
        _setUint(
            keccak256(abi.encodePacked("token.version", _name)),
            version
        );
    }
}
```

The **initialize** method allows to store token details after deployment. Only the contract owner can do that.

Suggestion:

- ⚠️ The storage values for “token.name”, “token.symbol”, “token.decimals” are never used because of the local version “_name”, “_symbol” and “_decimals” are. They are here for future purpose only as explained by the DeCash team.

```
function _getTotalSupply() internal view returns (uint256) {
    return
        _getUint(keccak256(abi.encodePacked("token.totalSupply", _name)));
}

function _setTotalSupply(uint256 _supply) internal {
    _setUint(
        keccak256(abi.encodePacked("token.totalSupply", _name)),
        _supply
    );
}

function _addTotalSupply(uint256 _supply) internal {
    _setTotalSupply(_getTotalSupply().add(_supply));
}

function _subTotalSupply(uint256 _supply) internal {
    _setTotalSupply(_getTotalSupply().sub(_supply));
}

function _getAllowed(address _owner, address _spender)
    internal
    view
    returns (uint256)
{
    return
        _getUint(
            keccak256(
                abi.encodePacked("token.allowed", _name, _owner, _spender)
            )
        );
}
```

```
function _setAllowed(
   address _owner,
   address _spender,
   uint256 _remaining
) internal {
   _setUint(
       keccak256(
           abi.encodePacked("token.allowed", _name, _owner, _spender)
       ),
       _remaining
   );
}

function _addAllowed(
   address _owner,
   address _spender,
   uint256 _balance
) internal {
   _setAllowed(
       _owner,
       _spender,
       _getAllowed(_owner, _spender).add(_balance)
   );
}

function _subAllowed(
   address _owner,
   address _spender,
   uint256 _balance
) internal {
   _setAllowed(
       _owner,
       _spender,
       _getAllowed(_owner, _spender).sub(_balance)
   );
}

function _getBalance(address _owner) internal view returns (uint256) {
   return
       _getUint(
           keccak256(abi.encodePacked("token.balance", _name, _owner))
       );
}

function _setBalance(address _owner, uint256 _balance) internal {
   require(!_isBlacklisted(_owner), "Blacklisted");
   _setUint(
       keccak256(abi.encodePacked("token.balance", _name, _owner)),
       _balance
   );
}

function _addBalance(address _owner, uint256 _balance) internal {
   _setBalance(_owner, _getBalance(_owner).add(_balance));
}

function _subBalance(address _owner, uint256 _balance) internal {
   _setBalance(_owner, _getBalance(_owner).sub(_balance));
}
```

```
function _getReqSign() internal view returns (uint256) {
   return _getUint(keccak256(abi.encodePacked("mint.reqSign", _name)));
}

function _getSignGeneration() internal view returns (uint256) {
   return _getUint(keccak256(abi.encodePacked("sign.generation", _name)));
}

function _getUsedSigIds(address _signer, uint256 _sigId)
   internal
   view
   returns (bool)
{
   return
       _getBool(
           keccak256(
               abi.encodePacked("sign.generation", _name, _signer, _sigId)
           )
       );
}

function _setReqSign(uint256 _reqsign) internal {
   _setUint(keccak256(abi.encodePacked("mint.reqSign", _name)), _reqsign);
}

function _setSignGeneration(uint256 _generation) internal {
   _setUint(
       keccak256(abi.encodePacked("sign.generation", _name)),
       _generation
   );
}

function _setUsedSigIds(
   address _signer,
   uint256 _sigId,
   bool _used
) internal {
   _setBool(
       keccak256(
           abi.encodePacked("sign.generation", _name, _signer, _sigId)
       ),
       _used
   );
}
```

These methods are internal functions to handle storage. These methods are used as utilities in the below functions.

```
function changeRequiredSigners(uint256 _reqsign)
   external
   onlySuperUser
   onlyLastest
   returns (uint256)
{
   _setReqSign(_reqsign);
```

```
    uint256 _generation = _getSignGeneration() + 1;
    _setSignGeneration(_generation);

    emit RequiredSignerChanged(_reqsign, _generation);

    return _generation;
}
```

The **changeRequiredSigners** allows owners to change the number of required signatures for multiSignature operations.

```
function name() external view returns (string memory) {
    return _name;
}

function symbol() external view returns (string memory) {
    return _symbol;
}

function decimals() external view returns (uint8) {
    return _decimals;
}

function totalSupply() external view returns (uint256) {
    return _getTotalSupply();
}

function balanceOf(address _owner)
    external
    view
    override
    returns (uint256)
{
    return _getBalance(_owner);
}

function allowance(address _owner, address _spender)
    external
    view
    override
    returns (uint256)
{
    return _getAllowed(_owner, _spender);
}

function transfer(address _to, uint256 _value)
    external
    override
    onlyLastest
    whenNotPaused
    returns (bool)
{
    return _transfer(msg.sender, _to, _value);
}

function transferFrom(
    address _from,
```

```
    address _to,
    uint256 _value
) external override onlyLastest whenNotPaused returns (bool) {
    return _transferFrom(msg.sender, _from, _to, _value);
}

function approve(address _spender, uint256 _value)
    external
    override
    onlyLastest
    whenNotPaused
    returns (bool)
{
    return _approve(msg.sender, _spender, _value);
}
```

These methods implement the standard ERC20 methods.

```
function isPaused() public view returns (bool) {
    return _getBool(keccak256(abi.encodePacked("contract.paused", _name)));
}

function pause() external onlySuperUser onlyLastest whenNotPaused {
    _setBool(keccak256(abi.encodePacked("contract.paused", _name)), true);
    emit Paused(msg.sender);
}

function unpause() external onlySuperUser onlyLastest whenPaused {
    _setBool(keccak256(abi.encodePacked("contract.paused", _name)), false);
    emit Unpaused(msg.sender);
}
```

These methods allow to check if the contract is paused, set to paused, set to not paused.
Read considerations below.

```
function transferMany(address[] calldata _tos, uint256[] calldata _values)
    external
    override
    onlyLastest
    whenNotPaused
    returns (bool)
{
    return _transferMany(msg.sender, _tos, _values);
}
```

This method allows to transfer tokens to many receivers at once.

```
function burn(uint256 _value)
    external
    override
    onlyLastest
    whenNotPaused
    returns (bool)
{
    return _burn(msg.sender, _value);
```

```
}

function burnFrom(address _from, uint256 _value)
   external
   override
   onlyLastest
   whenNotPaused
   returns (bool)
{
   _approve(_from, msg.sender, _getAllowed(_from, msg.sender).sub(_value));

   return _burn(_from, _value);
}
```

These methods allow to burn tokens from anyone own account or from another one who previously allowed a token amount to be spent.

```
function mint(address _to, uint256 _value)
   external
   override
   onlySuperUser
   onlyLastest
   whenNotPaused
   onlyMultiSignature(_getReqSign(), _getSignGeneration())
   returns (bool success)
{
   _addBalance(_to, _value);
   _addTotalSupply(_value);

   emit Transfer(address(0), _to, _value);

   return true;
}
```

This method allows super users to mint more tokens.

```
function _burn(address _from, uint256 _value) internal returns (bool) {
   _subBalance(_from, _value);
   _subTotalSupply(_value);

   emit Transfer(_from, address(0), _value);

   return true;
}
```

This method implements the burn behavior.

```
function _transfer(
   address _sender,
   address _to,
   uint256 _value
) internal returns (bool) {
   _subBalance(_sender, _value);
   _addBalance(_to, _value);
```

```
    emit Transfer(_sender, _to, _value);

    return true;
}
```

This method implements the transfer behavior.

```
function _transferMany(
    address _sender,
    address[] calldata _tos,
    uint256[] calldata _values
) internal returns (bool) {
    uint256 tosLen = _tos.length;

    require(tosLen == _values.length, "Wrong array parameter");
    require(tosLen <= 100, "Too many receiver");

    _subBalance(_sender, _calculateTotal(_values));

    for (uint8 x = 0; x < tosLen; x++) {
        _addBalance(_tos[x], _values[x]);

        emit Transfer(_sender, _tos[x], _values[x]);
    }

    return true;
}
```

This method implements the transferMany behavior.

```
function _transferFrom(
    address _sender,
    address _from,
    address _to,
    uint256 _value
) internal returns (bool) {
    _subAllowed(_from, _sender, _value);
    _subBalance(_from, _value);
    _addBalance(_to, _value);

    emit Transfer(_from, _to, _value);

    return true;
}
```

This method implements the transferFrom behavior.

```
function _approve(
    address _sender,
    address _spender,
    uint256 _value
) internal returns (bool) {
```

```
    _setAllowed(_sender, _spender, _value);

    emit Approval(_sender, _spender, _value);

    return true;
}
```

This method implements the approve behavior.

```
function _calculateTotal(uint256[] memory _values)
    internal
    pure
    returns (uint256)
{
    uint256 total = 0;
    uint256 ln = _values.length;

    for (uint8 x = 0; x < ln; x++) {
        total = total.add(_values[x]);
    }

    return total;
}
```

The **calculateTotal** method is a utility function to calculate the total amount of tokens to be transferred in a “transferMany” call.

```
function _validateViaSignatureParams(
    address _delegator,
    address _from,
    address _feeRecipient,
    uint256 _deadline,
    uint256 _sigId
) internal view {
    require(!isPaused(), "Contract paused");
    require(_isDelegator(_delegator), "Sender is not a delegator");
    require(_isFeeRecipient(_feeRecipient), "Invalid fee recipient");
    require(block.timestamp <= _deadline, "Request expired");
    require(!_getUsedSigIds(_from, _sigId), "Request already used");
}
```

This function is used to avoid the use of the modifier that can cause the “stack too deep” error.

Suggestion:

- ⚠️ Would be better to have this function as a modifier but it needs code rewriting in order to remove some variables or use structs to not have the “stack too deep” error.

```
function _burnSigId(address _from, uint256 _sigId) internal {
   _setUsedSigIds(_from, _sigId, true);
}
```

This function burns a signature Id after using.

```
function transferViaSignature(
   address _from,
   address _to,
   uint256 _value,
   uint256 _fee,
   address _feeRecipient,
   uint256 _deadline,
   uint256 _sigId,
   bytes calldata _sig,
   Signature.Std _sigStd
) external onlyLastest {
   _validateViaSignatureParams(
       msg.sender,
       _from,
       _feeRecipient,
       _deadline,
       _sigId
   );

   Signature.requireSignature(
       keccak256(
           abi.encodePacked(
               address(this),
               _from,
               _to,
               _value,
               _fee,
               _feeRecipient,
               _deadline,
               _sigId
           )
       ),
       _from,
       _sig,
       _sigStd,
       Signature.Dest.transfer
   );

   _subBalance(_from, _value.add(_fee)); // Subtract (value + fee)
   _addBalance(_to, _value);
   emit Transfer(_from, _to, _value);

   if (_fee > 0) {
       _addBalance(_feeRecipient, _fee);
       emit Transfer(_from, _feeRecipient, _fee);
   }

   _burnSigId(_from, _sigId);
}
```

This function distincts transaction signer from transaction executor. It allows anyone to transfer tokens from the `from` account by providing a valid signature, which can only be obtained from the `from` account owner. Note that passed parameter sigId is unique and cannot be passed twice (prevents replay attacks). When there's a need to make signature once again (because the first one is lost or whatever), user should sign the message with the same sigId, thus ensuring that the previous signature won't be used if the new one passes.
Use case: the user wants to send some tokens to another user or smart contract, but don't have ether to do so.

Params:

- **_from**: the account giving its signature to transfer `value` tokens to `to` address
- **_to**: the account receiving `value` tokens
- **_value**: the value in tokens to transfer
- **_fee**: a fee to pay to `feeRecipient`
- **_feeRecipient**: account which will receive fee
- **_deadline**: until when the signature is valid
- **_sigId**: signature unique ID. Signatures made with the same signature ID cannot be submitted twice
- **_sig**: signature made by `from`, which is the proof of `from`'s agreement with the above parameters
- **_sigStd**: chosen standard for signature validation. The signer must explicitly tell which standard they use

```
function transferManyViaSignature(
   address _from,
   address[] calldata _tos,
   uint256[] calldata _values,
   uint256 _fee,
   address _feeRecipient,
   uint256 _deadline,
   uint256 _sigId,
   bytes calldata _sig,
   Signature.Std _sigStd
) external onlyLastest {
   uint256 tosLen = _tos.length;

   require(tosLen == _values.length, "Wrong array parameters");
   require(tosLen <= 100, "Too many receiver");

   _validateViaSignatureParams(
       msg.sender,
       _from,
       _feeRecipient,
       _deadline,
       _sigId
   );

   bytes32 multisig = Signature.calculateManySig(_tos, _values);

   Signature.requireSignature(
       keccak256(
           abi.encodePacked(
               address(this),
               _from,
               multisig,
```

```
                _fee,
                _feeRecipient,
                _deadline,
                _sigId
            )
        ),
        _from,
        _sig,
        _sigStd,
        Signature.Dest.transferMany
    );

    _subBalance(_from, _calculateTotal(_values).add(_fee));

    for (uint8 x = 0; x < tosLen; x++) {
        _addBalance(_tos[x], _values[x]);
        emit Transfer(_from, _tos[x], _values[x]);
    }

    if (_fee > 0) {
        _addBalance(_feeRecipient, _fee);
        emit Transfer(_from, _feeRecipient, _fee);
    }

    _burnSigId(_from, _sigId);
}
```

This function distincts transaction signer from transaction executor. It allows anyone to transfer tokens from the `from` account to multiple recipient address by providing a valid signature, which can only be obtained from the `from` account owner. Note that passed parameter sigId is unique and cannot be passed twice (prevents replay attacks). When there's a need to make signature once again (because the first one is lost or whatever), user should sign the message with the same sigId, thus ensuring that the previous signature won't be used if the new one passes.
Use case: the user wants to send some tokens to multiple users or smart contracts, but don't have ether to do so.

Params:

- **_from**: the account giving its signature to transfer `value` tokens to `to` address
- **_tos[]**: array of account recipients
- **_values[]**: array of amount
- **_fee**: a fee to pay to `feeRecipient`
- **_feeRecipient**: account which will receive fee
- **_deadline**: until when the signature is valid
- **_sigId**: signature unique ID. Signatures made with the same signature ID cannot be submitted twice
- **_sig**: signature made by `from`, which is the proof of `from`'s agreement with the above parameters
- **_sigStd**: chosen standard for signature validation. The signer must explicitly tell which standard they use

```
function approveViaSignature(
    address _from,
    address _spender,
    uint256 _value,
    uint256 _fee,
```

```
    address _feeRecipient,
    uint256 _deadline,
    uint256 _sigId,
    bytes calldata _sig,
    Signature.Std _sigStd
) external onlyLastest {
    _validateViaSignatureParams(
        msg.sender,
        _from,
        _feeRecipient,
        _deadline,
        _sigId
    );

    Signature.requireSignature(
        keccak256(
            abi.encodePacked(
                address(this),
                _from,
                _spender,
                _value,
                _fee,
                _feeRecipient,
                _deadline,
                _sigId
            )
        ),
        _from,
        _sig,
        _sigStd,
        Signature.Dest.approve
    );

    if (_fee > 0) {
        _subBalance(_from, _fee);
        _addBalance(_feeRecipient, _fee);
        emit Transfer(_from, _feeRecipient, _fee);
    }

    _setAllowed(_from, _spender, _value);
    emit Approval(_from, _spender, _value);

    _burnSigId(_from, _sigId);
}
```

Same as `transferViaSignature`, but for `approve`.

Use case: the user wants to set an allowance for the smart contract or another user without having ether on their balance.

Params:

- **_from**: the account to approve withdrawal from, which signed all below parameters
- **_spender**: the account allowed to withdraw tokens from `from` address
- **_value**: the value in tokens to approve to withdraw
- **_fee**: a fee to pay to `feeRecipient`
- **_feeRecipient**: account which will receive fee

- **_deadline**: until when the signature is valid
- **_sigId**: signature unique ID. Signatures made with the same signature ID cannot be submitted twice
- **_sig**: signature made by `from`, which is the proof of `from`'s agreement with the above parameters
- **_sigStd**: chosen standard for signature validation. The signer must explicitly tell which standard they use

```
function transferFromViaSignature(
   address _signer,
   address _from,
   address _to,
   uint256 _value,
   uint256 _fee,
   address _feeRecipient,
   uint256 _deadline,
   uint256 _sigId,
   bytes calldata _sig,
   Signature.Std _sigStd
) external onlyLastest {
   _validateViaSignatureParams(
       msg.sender,
       _from,
       _feeRecipient,
       _deadline,
       _sigId
   );

   Signature.requireSignature(
       keccak256(
           abi.encodePacked(
               address(this),
               _from,
               _to,
               _value,
               _fee,
               _feeRecipient,
               _deadline,
               _sigId
           )
       ),
       _signer,
       _sig,
       _sigStd,
       Signature.Dest.transferFrom
   );

   _subAllowed(_from, _signer, _value.add(_fee));

   _subBalance(_from, _value.add(_fee)); // Subtract (value + fee)
   _addBalance(_to, _value);
   emit Transfer(_from, _to, _value);

   if (_fee > 0) {
       _addBalance(_feeRecipient, _fee);
       emit Transfer(_from, _feeRecipient, _fee);
   }

   _burnSigId(_from, _sigId);
}
```

Same as `transferViaSignature`, but for `transferFrom`.

Use case: the user wants to withdraw tokens from a smart contract or another user who allowed the user to do so. Important note: the fee is subtracted from the `value`, and `to` address receives `value - fee`.

Params:

- **_signer**: the address allowed to call transferFrom, which signed all below parameters
- **_from**: the account to make withdrawal from
- **_to**: the address of the recipient
- **_value**: the value in tokens to withdraw
- **_fee**: a fee to pay to `feeRecipient`
- **_feeRecipient**: account which will receive fee
- **_deadline**: until when the signature is valid
- **_sigId**: signature unique ID. Signatures made with the same signature ID cannot be submitted twice
- **_sig**: signature made by `from`, which is the proof of `from`'s agreement with the above parameters
- **_sigStd**: chosen standard for signature validation. The signer must explicitly tell which standard they use

Style Guide

Order of Layout

Layout contract elements in the following order:

1. Pragma statements
2. Import statements
3. Interfaces
4. Libraries
5. Contracts

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables
3. Events
4. Functions

Suggestion:

- ✅ Nothing.

Order of Functions

Ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier.

Functions should be grouped according to their visibility and ordered:

1. constructor
2. receive function (if exists)
3. fallback function (if exists)
4. external

5. public
6. internal
7. private

Suggestion:

- ✅ Nothing.

Function Declaration

For short function declarations, it is recommended for the opening brace of the function body to be kept on the same line as the function declaration.
The closing brace should be at the same indentation level as the function declaration.
The opening brace should be preceded by a single space.

The modifier order for a function should be:

1. Visibility
2. Mutability
3. Virtual
4. Override
5. Custom modifiers

For constructor functions on inherited contracts whose bases require arguments, it is recommended to drop the base constructors onto new lines in the same manner as modifiers if the function declaration is long or hard to read.

Suggestion:

- ⚠️ Code has been linted using prettier and the solidity plugin. It is an automatic code formatter that saves a lot of time formatting code. However it is an automatic tool so, depending on settings and on original code, it could be better to give a manual lint after a prettier application.

  For instance code like this

```
event Paused(address indexed from);
event Unpaused(address indexed from);
event Transfer(address indexed from, address indexed to, uint256 value);
event Approval(
   address indexed owner,
   address indexed spender,
   uint256 value
);
```

could be manually rewritten to

```
event Paused(address indexed from);
event Unpaused(address indexed from);
event Transfer(address indexed from, address indexed to, uint256 value);
event Approval(address indexed owner, address indexed spender, uint256 value);
```

to be more readable.

Naming Convention

Naming conventions are powerful when adopted and used broadly. The use of different conventions can convey significant meta information that would otherwise not be immediately available.
The naming recommendations given here are intended to improve the readability, and thus they are not rules, but rather guidelines to try and help convey the most information through the names of things.

Contract and Library Names:

1. Contracts and libraries should be named using the CapWords style. Examples: SimpleToken, SmartBank, CertificateHashRepository, Player, Congress, Owned.
2. Contract and library names should also match their filenames.
3. If a contract file includes multiple contracts and/or libraries, then the filename should match the core contract. This is not recommended however if it can be avoided.

Suggestion:

- ✅ Nothing.

Considerations

Overprivileged Audit:

There is no upper cap to token supply. It means that token owner/minters could generate more tokens than initially declared or backed by real assets. So if there is any public logic in token issuance it should be coded into the mint function otherwise holders or investors must be careful and they must rely on centralized token governance.
We do not consider this as a severity because of the following DeCash team's explanation:

`“The decision not to define a hard cap and therefore to opt for an unlimited supply was a strategic choice defined at the planning stage to avoid any possibility of significant appreciation of the token and therefore a reclassification as security.”`

Pausable behaviour**:** there could be many reasons to have this feature in an ERC20 token but a malicious owner/admin could try to set the token as *paused* and do not never unpause it. Token could remain in a stuck state and holders won't be able to recover/transfer their tokens.
We do not consider this as a severity because of the following DeCash team's explanation:

`“The company will draft suitable policies (contingency plan and crisis management plan) to manage, among other things, the triggering of the pause function.”`

Additional Informations

The DeCash team asked us to also check if DeCashToken will be compliant with Chainalysis requirements. Chainalysis is a blockchain analysis company that provides data, software, services, and research to government agencies, exchanges, financial institutions, and insurance and cybersecurity companies in over 50 countries. They added insights about ERC20s and they actually support 81 different ERC20 as described in this blog post.

They say:

```
[...]
ERC-20 is a technical standard so it has precise specifications for the functions and events of a
token smart contract as detailed above.
For example, the integral transfer(address _to, uint256 _value) function deducts the numerical
amount of the token equal to '_value' from the function caller's balance and adds it to the
balance controlled by the owner of Ethereum address  '_to'.
[...]
First off, unlike other EIPs that are embedded in the core blockchain protocol or APIs, smart
contract standards cannot be enforced. There's nothing stopping the coder of a token smart
contract picking and choosing the methods they want. You could even code a smart contract with
the exact same names as the ERC-20 methods but completely different, or even malicious,
functionalities.
Some popular tokens are only partially ERC-20 compliant. For example, the GNT Golem Network Token
does not implement the approve(...), allowance(..) and transferFrom(...) functions, or the
Approval(...) event.
Additionally, plenty of tokens have extra functionality on top of the ERC-20 specified methods.
These can significantly affect the flow of funds. For example MANA, Decentraland's token that
we've just added, has, Mint and Burn events. If you don't capture additional events correctly,
then funds will be appearing and disappearing all over the place and flows will be misleading or
outright incorrect.
[...]
Missing even tiny value transfers can cause massive issues as they accrete and have knock on
effects on addresses further down the flow of funds. This causes the inaccuracy in balances and
transfers of entities to balloon, resulting in completely unreliable data.
[...]
This is why we have a partially automated process for onboarding new ERC-20 tokens into our
systems. Every single token we support has been thoroughly investigated by our blockchain experts
to make sure we've captured every single nuance. We don't support a token unless we are sure you
can depend on our insights.
[...]
Since we can't automatically support every ERC-20, we need to ruthlessly prioritise which ones we
include in each of our quarterly batches.
Chainalysis' mission is to create transparency for a global economy built on blockchains.
Cryptocurrencies need greater trust and transparency to achieve their full transformative
potential. We contribute to this through our compliance and investigation tools as well as our
expert education and support.
To further our mission, we prioritise supporting the coins currently providing the most utility
to the most people. This way, we can be as effective as possible in providing value to not just
our own customers, but all actors in the cryptocurrency ecosystem. We can then accelerate the
growth and reach of cryptocurrency, helping all different kinds of people transact in new and
better ways.
[...]
```

So as any DeCashToken transfer emits a standard Transfer event, other than the mint, burn, burnFrom and also any approve emits an Approval event it SHOULD be compliant to the Chainalysis requirements.

Anyway this doesn't imply that DeCashToken(s) will be added into their products as they describe in their post, it is not an automated process.

We suggest contacting their customer support to ask for detailed information.

Conclusion

The DeCashToken is intended to be an upgradeable ERC20.
Usually raw ERC20s are pretty simple so having that number of code lines and features without any provided test might make the audit process hard.

The entire codebase is really complex as it uses a storage contract, a role manager contract, an upgrade manager contract, a proxy to delegate calls to the final ERC20 contract (also with signature transfers). Token also has a MultiSignature feature (better name could be MultiOwner) to handle minting.

It is mostly credited by the Rocket Pool approach, as their purpose is to build a network of contracts working together to handle vault, pool, deposits, etc. If the DeCash team's purpose is similar, this complex architecture might help in future adding of new features to the network. Otherwise there could be a simple way to write upgradeable ERC20 with delegated GAS payment.

Anyway the code as is can be extended for future purposes just upgrading any single contract, making attention on storage format and datas.

Before deploying live, we strongly recommend creating automated tests with a coverage percent near to 100%.