ULCDocumentsCore

Apr 10, 2019

Contents

ULCDocuments

The Blockchain-based signature app

Important: The documentation is not complete yet. It can contain (many) spelling mistakes and may change in near future.

The first goal of this documentation is to explain main mechanism of ULCDocuments in Ethereum blockchain. Then, everybody will be able to develop WebApp or software that is able to interact with ULCDocuments.

The source code of the project is available here

Note: For the beta version, full source code of *smart contracts* are not avaiable.

CHAPTER 1

What is ULCDocuments ?

ULCDocuments is an open-source software based on Ethereum blockchain. It gives to everybody a tool to sign all type of documents permanantly.

ULC is acronym of **U**ltra **L**ow **C**ost. The goal is to provide a very cheap way to use the blockchain, by providing free source code. ULCDocuments is a set of 2 Ethereum smart contract that has to be deployed by different actors to respect decentralized aspect. First, you need to pay once fees to be recorgnased by an authority. Then, you only pay Ethereum gaz and transaction fees.

CHAPTER 2

Why using ULCDocuments ?

Using ULCDocuments instead of simply send a transaction with your hash as data let the possibility for developers to **create tools or other smart contracts that can interact easier with your signatures**.

The aim of ULCDocuments is to be an open source project, used on many projects to **create an powerfull tool to get interaction between people' signatures.**

Then, everybody is welcome to create **extensions for PDF**, **Office/LibreOffice plugin**, **native applications** that work with ULCDocuments !

Moreover, feel free to create projects that include ULCDocuments verification in order to trust signature of a third party. You can use and/or create many JSON standards like Open Badges with ULCDocuments !

For the moment, we've created a WebApp that implement ULCDocumentsCore at https://ulcdocuments.blockchain-elite.fr/

CHAPTER 3

Who is behind the project ?

ULCDocuments is a project developped by Blockchain Elite Labs . The goal is to create some experimental tools based on Blockchain that are open-source, to promote Blockchain Elite know-how.

3.1 ULCDocKernel

3.1.1 Summary

Each organisation or individual has to publish his own **ULCDocKernel** smart contract. The kernel part is used to explicitly publish signatures to blockchain. So, the owner of ULCDocKernel has a complete and independent way to certify documents.

The kernel has different features :

- Multi-Owner ability
- Native timestamp tracking
- *Undestroyable* signatures

3.1.2 Multi-Owner and Multi-Operator ability

For most of important organization, only one person can't do dangerous actions alone, such as signing an official document. In this way, the kernel has the ability to wait for multiple **confirmations** before doing important stuff, like publishing a signature, or changing critical configuration parameters.

Roles

ULCDocKernel has 2 floors of administrative process :

- `owners` that are administrators of the Smart Contract. They can change sensible parameters.
- `operators` that can only push, confirm and revoke signatures.

Note: Owners have operators rights

In order to do something on the Smart Contract, like **signing** or **changing parameters**, you need to call every time a **requester** that will record you request and do it if it reaches `operatorsForChange` and respectively `ownersForChange` request count.

Warning: An account can't be `owner` and `operator` at the same time.

By default, ULCDocKernel is configurated to work with one owner account. So, is you use the kernel with only one owner, the request part is totally transparent and then they are done immediately.

How the requester works

When you want to do an action, all requesters will create a `keccak256` hash of the request and store it inside a mapping.

```
mapping(bytes32 => address[]) public doOwnerRequest;
mapping(bytes32 => address[]) public doOperatorRequest;
```

Then, each address who request the same thing will be added into an array and when it's length reaches `operatorsForChange` or `ownersForChange`, the action requested is done.

Constructor

```
constructor() public {
    owners[msg.sender] = true;
    ownerCount = 1;
    ownersForChange = 1;
    operatorsForChange = 1;
}
```

By default, the creator of the smart contract is the owner and everything is configurated to work with only one confirmation.

Variables available

```
uint256 public ownerCount;
uint256 public ownersForChange;

uint256 public operatorCount;
uint256 public operatorsForChange;

mapping (address => bool) public owners;
mapping(bytes32 => address[]) public doOwnerRequest;

mapping (address => bool) public operators;
mapping(bytes32 => address[]) public doOperatorRequest;

event OwnershipNewInteractor(address indexed newOwner);
```

(continues on next page)

(continued from previous page)

```
event OwnershipLossInteractor(address indexed oldOwner);

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

event OperatorNewInteractor(address indexed newOperator);

event OperatorLossInteractor(address indexed oldOperator);

event OperatorshipTransferred(
    address indexed previousOperator,
    address indexed newOperator
);
```

Functions available

Basic setters

```
//Both need enough confirmations (they are requester too)
function  setOperatorsForChange(uint256 _nb) external onlyOwner {}

function setOwnersForChange(uint256 _nb) external onlyOwner {}
```

Requester

```
function requestAddOwner(address _newOwner) external onlyOwner{}

function requestAddOperator(address _newOperator) external onlyOwner {}

function requestChangeOwner(address _oldOwner, address _newOwner) external onlyOwner{}

function requestRemoveOwner(address _removableOwner) external onlyOwner{}

function requestRemoveOperator(address _removableOperator) external onlyOwner{}
```

Getters

```
//Returns all adresses who approved the keccak256 operator request
function getOperatorRequest(bytes32 _theKey) external view returns(address[] memory)
↪{}

//Returns numbers of operators who confirmed the keccak256 request.
function getOperatorRequestLength(bytes32 _theKey) external view returns(uint256) {}

//Returns all adresses who approved the keccak256 owner request
function getOwnerRequest(bytes32 _theKey) external view returns(address[] memory) {}
```

(continues on next page)

(continued from previous page)

```
//Returns numbers of owners who confirmed the keccak256 request.
function getOwnerRequestLength(bytes32 _theKey) external view returns(uint256) {}
```

3.1.3 How signatures are stored

Signature Structure

ULCDocKernel has a struct called `Document` which has all information about a signed document.

```
struct Document {
      bool initialized;
      bool signed;
      bool revoked;

      uint256 signed_date;
      uint256 revoked_date;
      uint16 document_family;

      string revoked_reason;
      string source;
      string extra_data;
}
```

By default, the EVM makes all var set to `false`, `0`, or `""`.

- `initialized` is set to `true` as soon as someone started to try signing a document. When it is activated, you can't push an another version of the document. It's a security to prevent **deleting**, **cheating** about the fact that you sign some extra data, document family and so on.
- `signed` is set to `true` as long as the document has enough confirmations. **It's the only field you need to trust to know if something has been signed or not.**
- `revoked` is set to `true` as long as the document has enough revoke request from operators.
- `signed_date` is the UNIX timestamp (result of `block.timestamp` in solidity) of the block where signature has been officially signed.
- `revoked_date` is the UNIX timestamp (result of `block.timestamp` in solidity) of the block where revoked state has been officially declared.
- `document_family` is the index of the array where is stored the string value.
- `revoked_reason` is defined by operators when they push a revoke statement.
- `source` is defined by operators when they push document. It is the location where we can find the document if it's public.
- `extra_data` is defined by operators when they push document. It is a free location with `param:value, param2:value2`. It can be used to add extra information for automatic process, or to be compatible with newer Kernel Version.

Find a signature

```
mapping(bytes32 => Document) public Signatures_Book;
```

To find a signature, you need its bytes32 code. To obtain it, just check the Hash_Algorithm string. By default, ULCDocKernel uses **SHA3-256** hash of the document.

Note: Because mapping hashes its key with a *32 bytes* format, it is useless to use hash algorithm with more than *32 bytes* output like SHA3-512.

Constructor

```
constructor() public {
    Hash_Algorithm = "SHA3-256";
}
```

Variables available

```
uint256 public Document_Counter; // stat purposes only
string public Hash_Algorithm; //Essential Information to know how to hash files

string[] public document_family_registred = ["Undefined",
"Diploma",
"Bill",
"Order",
"Letter",
"Publication",
"Code",
"Image",
"Audio",
"Video",
"Binary"];

mapping(bytes32 => Document) public Signatures_Book;
```

Function List

Pushing something is the first step to do sign a document with data. Then, you need to confirm it before changing sign state to true.

```
function pushDocument(bytes32 _SignatureHash, string memory _source, uint16 _
↪indexDocumentFamily, string memory _extra_data) public atLeastOperator␣
↪whenNotPaused notUpgraded{}
```

Note: When you push a document into your Kernel, you automatically confirm it. So, if you use a simple signature Kernel, your document is signed with only one transaction.

```
//Request to confirm a signature. It can also be used to simply sign document␣
↪without extra_data.
function confirmDocument(bytes32 _SignatureHash) public atLeastOperator whenNotPaused␣
↪notUpgraded{}

//Request to add a "revoked" statement on the signature, and add a reason for that␣
↪(can be then displayed on clients).
```

(continues on next page)

(continued from previous page)

```
function pushRevokeDocument(bytes32 _SignatureHash, string calldata _reason) external␣
↪atLeastOperator whenNotPaused {}

//Request to confirm a revoke statement. It can also be used to simply revoke␣
↪document without reason
function confirmRevokeDocument(bytes32 _SignatureHash) external atLeastOperator␣
↪whenNotPaused {}
```

3.2 ULCDocMod

3.2.1 Summary

The aim of the *ULCDocMod* smart contract is to be an authority that certifies *ULCDocKernel* address. It has to guarantee the integrity of organisation's *ULCDocKernel* code and the identity of the owner. We can slightly compare it to a *Certificate Authority* for https protocol.

As everyone can publish a *ULCDocMod* smart contract, the acknowledgement is only based on the trust the user give. For example, Blockchain-Elite, the developper of ULCDocuments, publishes it's own ULCDocMod smart contract.

3.2.2 More in depth